



# Traffic Management

⌚ 22 minute read

---

Istio's traffic routing rules let you easily control the flow of traffic and API calls between services. Istio simplifies configuration of service-level properties like circuit breakers, timeouts, and retries, and makes it easy to set up important tasks like A/B testing, canary rollouts, and staged rollouts with percentage-based traffic splits. It also provides out-of-box failure

recovery features that help make your application more robust against failures of dependent services or the network.

Istio's traffic management model relies on the Envoy proxies that are deployed along with your services. All traffic that your mesh services send and receive (data plane traffic) is proxied through Envoy, making it easy to direct and control traffic around your mesh without making any changes to your services.

If you're interested in the details of how the features described in this guide work, you can find out more about Istio's traffic management implementation in

the architecture overview. The rest of this guide introduces Istio's traffic management features.

## **Introducing Istio traffic management**

In order to direct traffic within your mesh, Istio needs to know where all your endpoints are, and which services they belong to. To populate its own service registry, Istio connects to a service discovery system. For example, if you've installed Istio on a Kubernetes

cluster, then Istio automatically detects the services and endpoints in that cluster.

Using this service registry, the Envoy proxies can then direct traffic to the relevant services. Most microservice-based applications have multiple instances of each service workload to handle service traffic, sometimes referred to as a load balancing pool. By default, the Envoy proxies distribute traffic across each service's load balancing pool using a round-robin model, where requests are sent to each pool member in turn, returning to the top of the pool once each service instance has received a request.

While Istio's basic service discovery and load balancing gives you a working service mesh, it's far from all that Istio can do. In many cases you might want more fine-grained control over what happens to your mesh traffic. You might want to direct a particular percentage of traffic to a new version of a service as part of A/B testing, or apply a different load balancing policy to traffic for a particular subset of service instances. You might also want to apply special rules to traffic coming into or out of your mesh, or add an external dependency of your mesh to the service registry. You can do all this and more by adding your own traffic configuration to Istio using

## Istio's traffic management API.

Like other Istio configuration, the API is specified using Kubernetes custom resource definitions (CRDs), which you can configure using YAML, as you'll see in the examples.

The rest of this guide examines each of the traffic management API resources and what you can do with them. These resources are:

- Virtual services
- Destination rules
- Gateways

- Service entries
- Sidecars

This guide also gives an overview of some of the network resilience and testing features **that are built in to the API resources.**

## Virtual services

Virtual services, along with destination rules, are the key building blocks of Istio's traffic routing functionality.

A virtual service lets you configure how requests are routed to a service within an Istio service mesh, building on the basic connectivity and discovery provided by Istio and your platform. Each virtual service consists of a set of routing rules that are evaluated in order, letting Istio match each given request to the virtual service to a specific real destination within the mesh. Your mesh can require multiple virtual services or none depending on your use case.

## Why use virtual services?

Virtual services play a key role in making Istio's traffic management flexible and powerful. They do this by strongly decoupling where clients send their requests from the destination workloads that actually implement them. Virtual services also provide a rich way of specifying different traffic routing rules for sending traffic to those workloads.

Why is this so useful? Without virtual services, Envoy distributes traffic using round-robin load balancing between all service instances, as described in the introduction. You can improve this behavior with what you know about the workloads. For example, some might represent a different version. This can be

useful in A/B testing, where you might want to configure traffic routes based on percentages across different service versions, or to direct traffic from your internal users to a particular set of instances.

With a virtual service, you can specify traffic behavior for one or more hostnames. You use routing rules in the virtual service that tell Envoy how to send the virtual service's traffic to appropriate destinations. Route destinations can be versions of the same service or entirely different services.

A typical use case is to send traffic to different versions of a service, specified as service subsets.

Clients send requests to the virtual service host as if it was a single entity, and Envoy then routes the traffic to the different versions depending on the virtual service rules: for example, “20% of calls go to the new version” or “calls from these users go to version 2”. This allows you to, for instance, create a canary rollout where you gradually increase the percentage of traffic that’s sent to a new service version. The traffic routing is completely separate from the instance deployment, meaning that the number of instances implementing the new service version can scale up and down based on traffic load without referring to traffic routing at all. By contrast,

container orchestration platforms like Kubernetes only support traffic distribution based on instance scaling, which quickly becomes complex. You can read more about how virtual services help with canary deployments in Canary Deployments using Istio.

Virtual services also let you:

- Address multiple application services through a single virtual service. If your mesh uses Kubernetes, for example, you can configure a virtual service to handle all services in a specific namespace. Mapping a single virtual service to multiple “real” services is particularly useful in

facilitating turning a monolithic application into a composite service built out of distinct microservices without requiring the consumers of the service to adapt to the transition. Your routing rules can specify “calls to these URIs of monolith.com go to microservice A”, and so on. You can see how this works in one of our examples below.

- Configure traffic rules in combination with gateways to control ingress and egress traffic.

In some cases you also need to configure destination rules to use these features, as these are where you specify your service subsets. Specifying service

subsets and other destination-specific policies in a separate object lets you reuse these cleanly between virtual services. You can find out more about destination rules in the next section.

## **Virtual service example**

The following virtual service routes requests to different versions of a service depending on whether the request comes from a particular user.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
    - reviews
  http:
    - match:
        - headers:
            end-user:
              exact: jason
      route:
        - destination:
            host: reviews
            subset: v2
    - route:
        - destination:
            host: reviews
            subset: v3
```

# The hosts field

The hosts field lists the virtual service's hosts - in other words, the user-addressable destination or destinations that these routing rules apply to. This is the address or addresses the client uses when sending requests to the service.

```
hosts:  
- reviews
```

The virtual service hostname can be an IP address, a DNS name, or, depending on the platform, a short name (such as a Kubernetes service short name) that

resolves, implicitly or explicitly, to a fully qualified domain name (FQDN). You can also use wildcard ("\*") prefixes, letting you create a single set of routing rules for all matching services. Virtual service hosts don't actually have to be part of the Istio service registry, they are simply virtual destinations. This lets you model traffic for virtual hosts that don't have routable entries inside the mesh.

## Routing rules

The `http` section contains the virtual service's routing rules, describing match conditions and actions for

routing HTTP/1.1, HTTP2, and gRPC traffic sent to the destination(s) specified in the hosts field (you can also use `tcp` and `tls` sections to configure routing rules for TCP and unterminated TLS traffic). A routing rule consists of the destination where you want the traffic to go and zero or more match conditions, depending on your use case.

## Match condition

The first routing rule in the example has a condition and so begins with the `match` field. In this case you want this routing to apply to all requests from the user “jason”, so you use the `headers`, `end-user`, and `exact` fields to select the appropriate requests.

```
- match:  
  - headers:  
    end-user:  
      exact: jason
```

## Destination

The route section's `destination` field specifies the actual destination for traffic that matches this condition. Unlike the virtual service's host(s), the destination's host must be a real destination that exists in Istio's service registry or Envoy won't know where to send traffic to it. This can be a mesh service with proxies or a non-mesh service added using a

service entry. In this case we're running on Kubernetes and the host name is a Kubernetes service name:

```
route:  
- destination:  
  host: reviews  
  subset: v2
```

Note in this and the other examples on this page, we use a Kubernetes short name for the destination hosts for simplicity. When this rule is evaluated, Istio adds a domain suffix based on the namespace of the virtual service that contains the routing rule to get the fully qualified name for the host. Using short names in our

examples also means that you can copy and try them in any namespace you like.

Using short names like this only works if the destination hosts and the virtual service are actually in the same Kubernetes namespace. Because using the Kubernetes short name can result in misconfigurations, we recommend that you specify fully qualified host names in production environments.

The destination section also specifies which subset of this Kubernetes service you want requests that match this rule's conditions to go to, in this case the subset named v2. You'll see how you define a service subset in the section on destination rules below.

## Routing rule precedence

Routing rules are **evaluated in sequential order from top to bottom**, with the first rule in the virtual service definition being given highest priority. In this case you want anything that doesn't match the first routing rule to go to a default destination, specified in

the second rule. Because of this, the second rule has no match conditions and just directs traffic to the v3 subset.

- route:
- destination:
  - host: reviews
  - subset: v3

We recommend providing a default “no condition” or weight-based rule (described below) like this as the last rule in each virtual service to ensure that traffic to the virtual service always has at least one matching route.

# More about routing rules

As you saw above, routing rules are a powerful tool for routing particular subsets of traffic to particular destinations. You can set match conditions on traffic ports, header fields, URIs, and more. For example, this virtual service lets users send traffic to two separate services, ratings and reviews, as if they were part of a bigger virtual service at <http://bookinfo.com/>. The virtual service rules match traffic based on request URIs and direct requests to the appropriate service.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: bookinfo
spec:
  hosts:
    - bookinfo.com
  http:
    - match:
        - uri:
            prefix: /reviews
        route:
          - destination:
              host: reviews
    - match:
        - uri:
            prefix: /ratings
        route:
          - destination:
              host: ratings
```

For some match conditions, you can also choose to select them using the exact value, a prefix, or a regex.

You can add multiple match conditions to the same match block to AND your conditions, or add multiple match blocks to the same rule to OR your conditions.

You can also have multiple routing rules for any given virtual service. This lets you make your routing conditions as complex or simple as you like within a single virtual service. A full list of match condition fields and their possible values can be found in the `HTTPMatchRequest` reference.

In addition to using match conditions, you can

distribute traffic by percentage “weight”. This is useful for A/B testing and canary rollouts:

```
spec:  
  hosts:  
    - reviews  
  http:  
    - route:  
        - destination:  
            host: reviews  
            subset: v1  
            weight: 75  
        - destination:  
            host: reviews  
            subset: v2  
            weight: 25
```

You can also use routing rules to perform some

actions on the traffic, for example:

- Append or remove headers.
- Rewrite the URL.
- Set a retry policy for calls to this destination.

To learn more about the actions available, see the [HTTPRoute](#) reference.

## Destination rules

Along with virtual services, destination rules are a key part of Istio's traffic routing functionality. You can think of virtual services as how you route your traffic **to** a given destination, and then you use destination rules to configure what happens to traffic **for** that destination. Destination rules are applied after virtual service routing rules are evaluated, so they apply to the traffic's "real" destination.

In particular, you use destination rules to specify named service subsets, such as grouping all a given service's instances by version. You can then use these service subsets in the routing rules of virtual services to control the traffic to different instances of your

services.

Destination rules also let you customize Envoy's traffic policies when calling the entire destination service or a particular service subset, such as your preferred load balancing model, TLS security mode, or circuit breaker settings. You can see a complete list of destination rule options in the [Destination Rule](#) reference.

## Load balancing options

By default, Istio uses a round-robin load balancing policy, where each service instance in the instance pool gets a request in turn. Istio also supports the following models, which you can specify in destination rules for requests to a particular service or service subset.

- Random: Requests are forwarded at random to instances in the pool.
- Weighted: Requests are forwarded to instances in the pool according to a specific percentage.
- Least requests: Requests are forwarded to instances with the least number of requests.

See the Envoy load balancing documentation for more information about each option.

## Destination rule example

The following example destination rule configures three different subsets for the `my-svc` destination service, with different load balancing policies:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: my-destination-rule
```

```
spec:
```

```
  host: my-svc
```

```
  trafficPolicy:
```

```
    loadBalancer:
```

```
      simple: RANDOM
```

```
subsets:
```

```
- name: v1
```

```
  labels:
```

```
    version: v1
```

```
- name: v2
```

```
  labels:
```

```
    version: v2
```

```
  trafficPolicy:
```

```
    loadBalancer:
```

```
      simple: ROUND_ROBIN
```

```
- name: v3
```

```
  labels:
```

```
    version: v3
```

Each subset is defined based on one or more labels, which in Kubernetes are key/value pairs that are attached to objects such as Pods. These labels are applied in the Kubernetes service's deployment as metadata to identify different versions.

As well as defining subsets, this destination rule has both a default traffic policy for all subsets in this destination and a subset-specific policy that overrides it for just that subset. The default policy, defined above the `subsets` field, sets a simple random load balancer for the `v1` and `v3` subsets. In the `v2` policy, a round-robin load balancer is specified in the corresponding subset's field.

# Gateways

You use a gateway to manage inbound and outbound traffic for your mesh, letting you specify which traffic you want to enter or leave the mesh. Gateway configurations are applied to standalone Envoy proxies that are running at the edge of the mesh, rather than sidecar Envoy proxies running alongside your service workloads.

Unlike other mechanisms for controlling traffic entering your systems, such as the Kubernetes Ingress APIs, Istio gateways let you use the full power

and flexibility of Istio's traffic routing. You can do this because Istio's Gateway resource just lets you configure layer 4-6 load balancing properties such as ports to expose, TLS settings, and so on. Then instead of adding application-layer traffic routing (L7) to the same API resource, you bind a regular Istio virtual service to the gateway. This lets you basically manage gateway traffic like any other data plane traffic in an Istio mesh.

Gateways are primarily used to manage ingress traffic, but you can also configure egress gateways. An egress gateway lets you configure a dedicated exit node for the traffic leaving the mesh, letting you limit

which services can or should access external networks, or to enable secure control of egress traffic to add security to your mesh, for example. You can also use a gateway to configure a purely internal proxy.

Istio provides some preconfigured gateway proxy deployments (`istio-ingressgateway` and `istio-egressgateway`) that you can use - both are deployed if you use our demo installation, while just the ingress gateway is deployed with our default profile. You can apply your own gateway configurations to these deployments or deploy and configure your own gateway proxies.

# Gateway example

The following example shows a possible gateway configuration for external HTTPS ingress traffic:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: ext-host-gwy
spec:
  selector:
    app: my-gateway-controller
  servers:
  - port:
      number: 443
      name: https
      protocol: HTTPS
    hosts:
    - ext-host.example.com
    tls:
      mode: SIMPLE
      credentialName: ext-host-cert
```

This gateway configuration lets HTTPS traffic from

`ext-host.example.com` into the mesh on port 443, but doesn't specify any routing for the traffic.

To specify routing and for the gateway to work as intended, you must also bind the gateway to a virtual service. You do this using the virtual service's `gateways` field, as shown in the following example:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: virtual-svc
spec:
  hosts:
    - ext-host.example.com
  gateways:
    - ext-host-gwy
```

You can then configure the virtual service with routing rules for the external traffic.

# Service entries

You use a service entry to add an entry to the service registry that Istio maintains internally. After you add the service entry, the Envoy proxies can send traffic to the service as if it was a service in your mesh. Configuring service entries allows you to manage traffic for services running outside of the mesh, including the following tasks:

- Redirect and forward traffic for external destinations, such as APIs consumed from the web, or traffic to services in legacy infrastructure.
- Define retry, timeout, and fault injection policies for external destinations.

- Run a mesh service in a Virtual Machine (VM) by adding VMs to your mesh.

You don't need to add a service entry for every external service that you want your mesh services to use. By default, Istio configures the Envoy proxies to passthrough requests to unknown services. However, you can't use Istio features to control the traffic to destinations that aren't registered in the mesh.

## Service entry example

The following example mesh-external service entry adds the ext-svc.example.com external dependency to Istio's service registry:

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: svc-entry
spec:
  hosts:
  - ext-svc.example.com
  ports:
  - number: 443
    name: https
    protocol: HTTPS
  location: MESH_EXTERNAL
  resolution: DNS
```

You specify the external resource using the `hosts` field. You can qualify it fully or use a wildcard prefixed domain name.

You can configure virtual services and destination rules to control traffic to a service entry in a more granular way, in the same way you configure traffic for any other service in the mesh. For example, the following destination rule configures the traffic route to use mutual TLS to secure the connection to the `ext-svc.example.com` external service that we configured using the service entry:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: ext-res-dr
spec:
  host: ext-svc.example.com
  trafficPolicy:
    tls:
      mode: MUTUAL
      clientCertificate: /etc/certs/myclientcert.pem
      privateKey: /etc/certs/client_private_key.pem
      caCertificates: /etc/certs/rootcacerts.pem
```

See the Service Entry reference for more possible configuration options.

# Sidecars

By default, Istio configures every Envoy proxy to accept traffic on all the ports of its associated workload, and to reach every workload in the mesh when forwarding traffic. You can use a sidecar configuration to do the following:

- Fine-tune the set of ports and protocols that an Envoy proxy accepts.
- Limit the set of services that the Envoy proxy can reach.

You might want to limit sidecar reachability like this in larger applications, where having every proxy configured to reach every other service in the mesh can potentially affect mesh performance due to high memory usage.

You can specify that you want a sidecar configuration to apply to all workloads in a particular namespace, or choose specific workloads using a `workloadSelector`. For example, the following sidecar configuration configures all services in the `bookinfo` namespace to only reach services running in the same namespace and the Istio control plane (currently needed to use Istio's policy and telemetry features):

```
apiVersion: networking.istio.io/v1alpha3
kind: Sidecar
metadata:
  name: default
  namespace: bookinfo
spec:
  egress:
    - hosts:
      - "./*"
      - "istio-system/*"
```

See the [Sidecar](#) reference for more details.

# Network resilience and

# testing

As well as helping you direct traffic around your mesh, Istio provides opt-in failure recovery and fault injection features that you can configure dynamically at runtime. Using these features helps your applications operate reliably, ensuring that the service mesh can tolerate failing nodes and preventing localized failures from cascading to other nodes.

# Timeouts

A timeout is the amount of time that an Envoy proxy should wait for replies from a given service, ensuring that services don't hang around waiting for replies indefinitely and that calls succeed or fail within a predictable timeframe. The Envoy timeout for HTTP requests is disabled in Istio by default.

For some applications and services, Istio's default timeout might not be appropriate. For example, a timeout that is too long could result in excessive latency from waiting for replies from failing services,

while a timeout that is too short could result in calls failing unnecessarily while waiting for an operation involving multiple services to return. To find and use your optimal timeout settings, Istio lets you easily adjust timeouts dynamically on a per-service basis using virtual services without having to edit your service code. Here's a virtual service that specifies a 10 second timeout for calls to the v1 subset of the ratings service:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
    - ratings
  http:
    - route:
        - destination:
            host: ratings
            subset: v1
        timeout: 10s
```

# Retries

A retry setting specifies the maximum number of times an Envoy proxy attempts to connect to a service if the initial call fails. Retries can enhance service availability and application performance by making sure that calls don't fail permanently because of transient problems such as a temporarily overloaded service or network. The interval between retries (25ms+) is variable and determined automatically by Istio, preventing the called service from being overwhelmed with requests. The default retry behavior for HTTP requests is to retry twice before returning the error.

Like timeouts, Istio's default retry behavior might not

suit your application needs in terms of latency (too many retries to a failed service can slow things down) or availability. Also like timeouts, you can adjust your retry settings on a per-service basis in virtual services without having to touch your service code. You can also further refine your retry behavior by adding per-retry timeouts, specifying the amount of time you want to wait for each retry attempt to successfully connect to the service. The following example configures a maximum of 3 retries to connect to this service subset after an initial call failure, each with a 2 second timeout.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
    - ratings
  http:
    - route:
        - destination:
            host: ratings
            subset: v1
  retries:
    attempts: 3
    perTryTimeout: 2s
```

# Circuit breakers

Circuit breakers are another useful mechanism Istio provides for creating resilient microservice-based applications. In a circuit breaker, you set limits for calls to individual hosts within a service, such as the number of concurrent connections or how many times calls to this host have failed. Once that limit has been reached the circuit breaker “trips” and stops further connections to that host. Using a circuit breaker pattern enables fast failure rather than clients trying to connect to an overloaded or failing host.

As circuit breaking applies to “real” mesh destinations in a load balancing pool, you configure circuit breaker thresholds in destination rules, with the

settings applying to each individual host in the service. The following example limits the number of concurrent connections for the `reviews` service workloads of the `v1` subset to 100:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: reviews
spec:
  host: reviews
  subsets:
  - name: v1
    labels:
      version: v1
    trafficPolicy:
      connectionPool:
        tcp:
          maxConnections: 100
```

You can find out more about creating circuit breakers  
in Circuit Breaking.

# Fault injection

After you've configured your network, including failure recovery policies, you can use Istio's fault injection mechanisms to test the failure recovery capacity of your application as a whole. Fault injection is a testing method that introduces errors into a system to ensure that it can withstand and recover from error conditions. Using fault injection can be particularly useful to ensure that your failure recovery policies aren't incompatible or too restrictive, potentially resulting in critical services being unavailable.

Unlike other mechanisms for introducing errors such as delaying packets or killing pods at the network layer, Istio lets you inject faults at the application layer. This lets you inject more relevant failures, such as HTTP error codes, to get more relevant results.

You can inject two types of faults, both configured using a virtual service:

- Delays: Delays are timing failures. They mimic increased network latency or an overloaded upstream service.
- Aborts: Aborts are crash failures. They mimic failures in upstream services. Aborts usually

manifest in the form of HTTP error codes or TCP connection failures.

For example, this virtual service introduces a 5 second delay for 1 out of every 1000 requests to the ratings **service**.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
    - ratings
  http:
    - fault:
        delay:
          percentage:
            value: 0.1
          fixedDelay: 5s
      route:
        - destination:
            host: ratings
            subset: v1
```

For detailed instructions on how to configure delays

and aborts, see Fault Injection.

# Working with your applications

Istio failure recovery features are completely transparent to the application. Applications don't know if an Envoy sidecar proxy is handling failures for a called service before returning a response. This means that if you are also setting failure recovery policies in your application code you need to keep in mind that both work independently, and therefore

might conflict. For example, suppose you can have two timeouts, one configured in a virtual service and another in the application. The application sets a 2 second timeout for an API call to a service. However, you configured a 3 second timeout with 1 retry in your virtual service. In this case, the application's timeout kicks in first, so your Envoy timeout and retry attempt has no effect.

While Istio failure recovery features improve the reliability and availability of services in the mesh, applications must handle the failure or errors and take appropriate fallback actions. For example, when all instances in a load balancing pool have failed,

Envoy returns an HTTP 503 code. The application must implement any fallback logic needed to handle the HTTP 503 error code.



# Security

⌚ 23 minute read

---

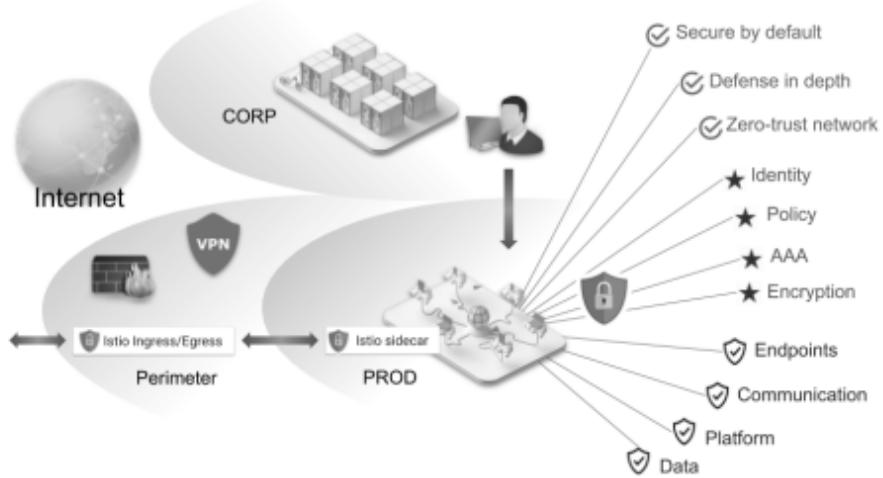
Breaking down a monolithic application into atomic services offers various benefits, including better agility, better scalability and better ability to reuse services. However, microservices also have particular security needs:

- To defend against man-in-the-middle attacks, they

need traffic encryption.

- To provide flexible service access control, they need mutual TLS and fine-grained access policies.
- To determine who did what at what time, they need auditing tools.

Istio Security provides a comprehensive security solution to solve these issues. This page gives an overview on how you can use Istio security features to secure your services, wherever you run them. In particular, Istio security mitigates both insider and external threats against your data, endpoints, communication, and platform.



## Security overview

The Istio security features provide strong identity, powerful policy, transparent TLS encryption, and authentication, authorization and audit (AAA) tools to

protect your services and data. The goals of Istio security are:

- Security by default: no changes needed to application code and infrastructure
- Defense in depth: integrate with existing security systems to provide multiple layers of defense
- Zero-trust network: build security solutions on distrusted networks

Visit our mutual TLS Migration docs to start using Istio security features with your deployed services. Visit our Security Tasks for detailed instructions to use the

security features.

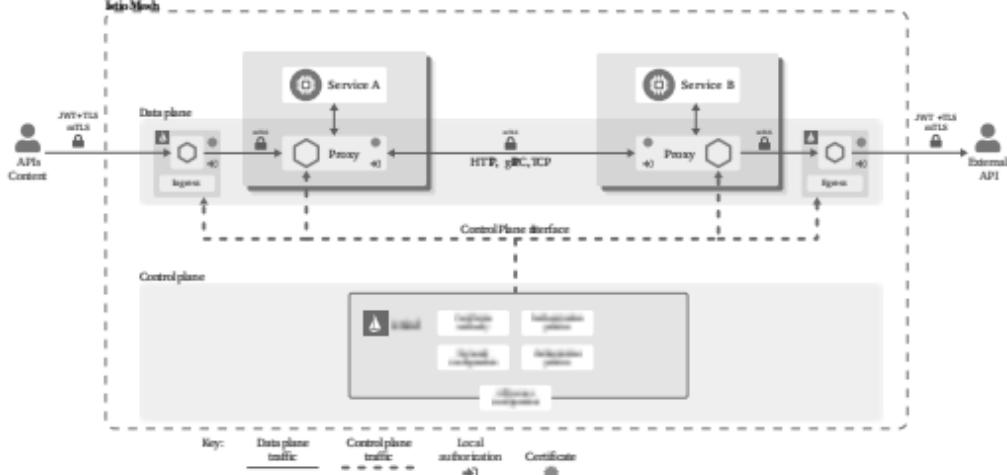
## High-level architecture

Security in Istio involves multiple components:

- A Certificate Authority (CA) for key and certificate management
- The configuration API server distributes to the proxies:
  - authentication policies

- authorization policies
- secure naming information
- Sidecar and perimeter proxies work as Policy Enforcement Points (PEPs) to secure communication between clients and servers.
- A set of Envoy proxy extensions to manage telemetry and auditing

The control plane handles configuration from the API server and configures the PEPs in the data plane. The PEPs are implemented using Envoy. The following diagram shows the architecture.



## Security Architecture

In the following sections, we introduce the Istio security features in detail.

# Istio identity

Identity is a fundamental concept of any security infrastructure. At the beginning of a workload-to-workload communication, the two parties must exchange credentials with their identity information for mutual authentication purposes. On the client side, the server's identity is checked against the secure naming information to see if it is an authorized runner of the workload. On the server side, the server can determine what information the client can access based on the authorization policies, audit who accessed what at what time, charge clients based on the

workloads they used, and reject any clients who failed to pay their bill from accessing the workloads.

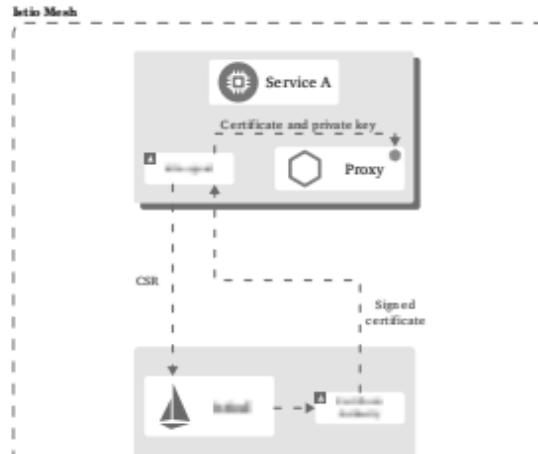
The Istio identity model uses the first-class `service identity` to determine the identity of a request's origin. This model allows for great flexibility and granularity for service identities to represent a human user, an individual workload, or a group of workloads. On platforms without a service identity, Istio can use other identities that can group workload instances, such as service names.

The following list shows examples of service identities that you can use on different platforms:

- Kubernetes: Kubernetes service account
- GCE: GCP service account
- On-premises (non-Kubernetes): user account, custom service account, service name, Istio service account, or GCP service account. The custom service account refers to the existing service account just like the identities that the customer's Identity Directory manages.

## **Identity and certificate management**

Istio securely provisions strong identities to every workload with X.509 certificates. Istio agents, running alongside each Envoy proxy, work together with `istiod` to automate key and certificate rotation at scale. The following diagram shows the identity provisioning flow.



# Identity Provisioning Workflow

Istio provisions keys and certificates through the following flow:

1. `istiod` offers a gRPC service to take certificate signing requests (CSRs).
2. When started, the Istio agent creates the private key and CSR, and then sends the CSR with its credentials to `istiod` for signing.

3. The CA in `istiod` validates the credentials carried in the CSR. Upon successful validation, it signs the CSR to generate the certificate.
4. When a workload is started, Envoy requests the certificate and key from the Istio agent in the same container via the Envoy secret discovery service (SDS) API.
5. The Istio agent sends the certificates received from `istiod` and the private key to Envoy via the Envoy SDS API.
6. Istio agent monitors the expiration of the workload certificate. The above process repeats periodically for certificate and key rotation.

# Authentication

Istio provides two types of authentication:

- Peer authentication: used for service-to-service authentication to verify the client making the connection. Istio offers mutual TLS as a full stack solution for transport authentication, which can be enabled without requiring service code changes. This solution:
  - Provides each service with a strong identity representing its role to enable interoperability across clusters and clouds.

- Secures service-to-service communication.
- Provides a key management system to automate key and certificate generation, distribution, and rotation.
- Request authentication: Used for end-user authentication to verify the credential attached to the request. Istio enables request-level authentication with JSON Web Token (JWT) validation and a streamlined developer experience using a custom authentication provider or any OpenID Connect providers, for example:
  - ORY Hydra

- Keycloak
- Auth0
- Firebase Auth
- Google Auth

In all cases, Istio stores the authentication policies in the Istio config store via a custom Kubernetes API. Istiod keeps them up-to-date for each proxy, along with the keys where appropriate. Additionally, Istio supports authentication in permissive mode to help you understand how a policy change can affect your security posture before it is enforced.

# Mutual TLS authentication

Istio tunnels service-to-service communication through the client- and server-side PEPs, which are implemented as Envoy proxies. When a workload sends a request to another workload using mutual TLS authentication, the request is handled as follows:

1. Istio re-routes the outbound traffic from a client to the client's local sidecar Envoy.
2. The client side Envoy starts a mutual TLS handshake with the server side Envoy. During the handshake, the client side Envoy also does a

secure naming check to verify that the service account presented in the server certificate is authorized to run the target service.

3. The client side Envoy and the server side Envoy establish a mutual TLS connection, and Istio forwards the traffic from the client side Envoy to the server side Envoy.
4. The server side Envoy authorizes the request. If authorized, it forwards the traffic to the backend service through local TCP connections.

Istio configures `TLSv1_2` as the minimum TLS version for both client and server with the following cipher

suites:

- ECDHE-ECDSA-AES256-GCM-SHA384
- ECDHE-RSA-AES256-GCM-SHA384
- ECDHE-ECDSA-AES128-GCM-SHA256
- ECDHE-RSA-AES128-GCM-SHA256
- AES256-GCM-SHA384
- AES128-GCM-SHA256

## Permissive mode

Istio mutual TLS has a permissive mode, which allows

a service to accept both plaintext traffic and mutual TLS traffic at the same time. This feature greatly improves the mutual TLS onboarding experience.

Many non-Istio clients communicating with a non-Istio server presents a problem for an operator who wants to migrate that server to Istio with mutual TLS enabled. Commonly, the operator cannot install an Istio sidecar for all clients at the same time or does not even have the permissions to do so on some clients. Even after installing the Istio sidecar on the server, the operator cannot enable mutual TLS without breaking existing communications.

With the permissive mode enabled, the server accepts both plaintext and mutual TLS traffic. The mode provides greater flexibility for the on-boarding process. The server's installed Istio sidecar takes mutual TLS traffic immediately without breaking existing plaintext traffic. As a result, the operator can gradually install and configure the client's Istio sidecars to send mutual TLS traffic. Once the configuration of the clients is complete, the operator can configure the server to mutual TLS only mode.

For more information, visit the [Mutual TLS Migration tutorial](#).

# Secure naming

Server identities are encoded in certificates, but service names are retrieved through the discovery service or DNS. The secure naming information maps the server identities to the service names. A mapping of identity  $A$  to service name  $B$  means “ $A$  is authorized to run service  $B$ ”. The control plane watches the apiserver, generates the secure naming mappings, and distributes them securely to the PEPs. The following example explains why secure naming is critical in authentication.

Suppose the legitimate servers that run the service datastore only use the `infra-team` identity. A malicious user has the certificate and key for the `test-team` identity. The malicious user intends to impersonate the service to inspect the data sent from the clients. The malicious user deploys a forged server with the certificate and key for the `test-team` identity. Suppose the malicious user successfully hijacked (through DNS spoofing, BGP/route hijacking, ARP spoofing, etc.) the traffic sent to the `datastore` and redirected it to the forged server.

When a client calls the `datastore` service, it extracts the `test-team` identity from the server's certificate, and

checks whether `test-team` is allowed to run `datastore` with the secure naming information. The client detects that `test-team` is not allowed to run the `datastore` service and the authentication fails.

Note that, for non HTTP/HTTPS traffic, secure naming doesn't protect from DNS spoofing, in which case the attacker modifies the destination IPs for the service. Since TCP traffic does not contain `Host` information and Envoy can only rely on the destination IP for routing, Envoy may route traffic to services on the hijacked IPs. This DNS spoofing can happen even before the client-side Envoy receives the traffic.

# Authentication architecture

You can specify authentication requirements for workloads receiving requests in an Istio mesh using peer and request authentication policies. The mesh operator uses `.yaml` files to specify the policies. The policies are saved in the Istio configuration storage once deployed. The Istio controller watches the configuration storage.

Upon any policy changes, the new policy is translated to the appropriate configuration telling the PEP how to perform the required authentication mechanisms.

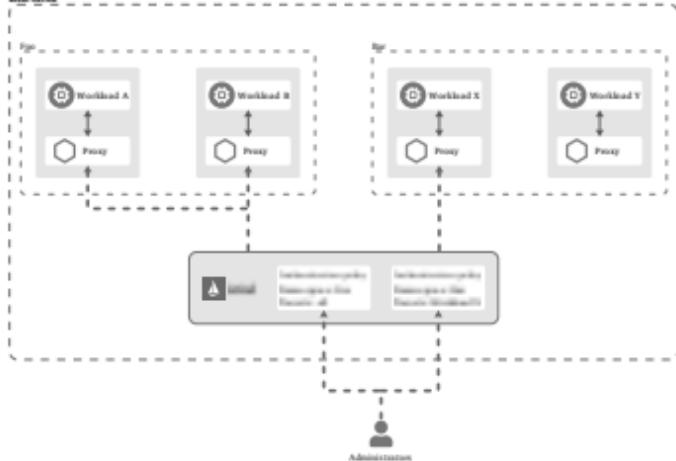
The control plane may fetch the public key and attach it to the configuration for JWT validation.

Alternatively, Istiod provides the path to the keys and certificates the Istio system manages and installs them to the application pod for mutual TLS. You can find more info in the [Identity and certificate management](#) section.

Istio sends configurations to the targeted endpoints asynchronously. Once the proxy receives the configuration, the new authentication requirement takes effect immediately on that pod.

Client services, those that send requests, are

responsible for following the necessary authentication mechanism. For request authentication, the application is responsible for acquiring and attaching the JWT credential to the request. For peer authentication, Istio automatically upgrades all traffic between two PEPs to mutual TLS. If authentication policies disable mutual TLS mode, Istio continues to use plain text between PEPs. To override this behavior explicitly disable mutual TLS mode with destination rules. You can find out more about how **mutual TLS works in the** Mutual TLS authentication section.



## Authentication Architecture

Istio outputs identities with both types of authentication, as well as other claims in the credential if applicable, to the next layer: authorization.

# Authentication policies

This section provides more details about how Istio authentication policies work. As you'll remember from the Architecture section, authentication policies apply to requests that a service receives. To specify client-side authentication rules in mutual TLS, you need to specify the `TLSSettings` in the `DestinationRule`. You can find more information in our TLS settings reference docs.

Like other Istio configurations, you can specify authentication policies in `.yaml` files. You deploy

policies using `kubectl`. The following example authentication policy specifies that transport authentication for the workloads with the `app:reviews` label must use mutual TLS:

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: "example-peer-policy"
  namespace: "foo"
spec:
  selector:
    matchLabels:
      app: reviews
  mtls:
    mode: STRICT
```

# Policy storage

Istio stores mesh-scope policies in the root namespace. These policies have an empty selector apply to all workloads in the mesh. Policies that have a namespace scope are stored in the corresponding namespace. They only apply to workloads within their namespace. If you configure a `selector` field, the authentication policy only applies to workloads matching the conditions you configured.

Peer and request authentication policies are stored separately by kind, `PeerAuthentication` and

`RequestAuthentication` respectively.

## Selector field

Peer and request authentication policies use selector fields to specify the label of the workloads to which the policy applies. The following example shows the selector field of a policy that applies to workloads with the `app:product-page` label:

```
selector:  
  matchLabels:  
    app: product-page
```

If you don't provide a value for the `selector` field, Istio matches the policy to all workloads in the storage scope of the policy. Thus, the `selector` fields help you specify the scope of the policies:

- Mesh-wide policy: A policy specified for the root namespace without or with an empty `selector` field.
- Namespace-wide policy: A policy specified for a non-root namespace without or with an empty `selector` field.
- Workload-specific policy: a policy defined in the regular namespace, with non-empty `selector` field.

Peer and request authentication policies follow the same hierarchy principles for the selector fields, but Istio combines and applies them in slightly different ways.

There can be only one mesh-wide peer authentication policy, and only one namespace-wide peer authentication policy per namespace. When you configure multiple mesh- or namespace-wide peer authentication policies for the same mesh or namespace, Istio ignores the newer policies. When more than one workload-specific peer authentication policy matches, Istio picks the oldest one.

Istio applies the narrowest matching policy for each workload using the following order:

1. workload-specific
2. namespace-wide
3. mesh-wide

Istio can combine all matching request authentication policies to work as if they come from a single request authentication policy. Thus, you can have multiple mesh-wide or namespace-wide policies in a mesh or namespace. However, it is still a good practice to avoid having multiple mesh-wide or namespace-wide

request authentication policies.

## Peer authentication

Peer authentication policies specify the mutual TLS mode Istio enforces on target workloads. The following modes are supported:

- **PERMISSIVE**: Workloads accept both mutual TLS and plain text traffic. This mode is most useful during migrations when workloads without sidecar cannot use mutual TLS. Once workloads are migrated with sidecar injection, you should switch the mode to **STRICT**.

- STRICT: Workloads only accept mutual TLS traffic.
- DISABLE: Mutual TLS is disabled. From a security perspective, you shouldn't use this mode unless you provide your own security solution.

When the mode is unset, the mode of the parent scope is inherited. Mesh-wide peer authentication policies with an unset mode use the PERMISSIVE mode by default.

The following peer authentication policy requires all workloads in namespace `foo` to use mutual TLS:

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: "example-policy"
  namespace: "foo"
spec:
  mtls:
    mode: STRICT
```

With workload-specific peer authentication policies, you can specify different mutual TLS modes for different ports. You can only use ports that workloads have claimed for port-wide mutual TLS configuration. The following example disables mutual TLS on port 80 for the `app:example-app` workload, and uses the mutual TLS settings of the namespace-wide peer

authentication policy for all other ports:

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: "example-workload-policy"
  namespace: "foo"
spec:
  selector:
    matchLabels:
      app: example-app
  portLevelMtls:
    80:
      mode: DISABLE
```

The peer authentication policy above works only because the service configuration below bound the

requests from the example-app workload to port 80 of the example-service:

```
apiVersion: v1
kind: Service
metadata:
  name: example-service
  namespace: foo
spec:
  ports:
  - name: http
    port: 8000
    protocol: TCP
    targetPort: 80
  selector:
    app: example-app
```

# Request authentication

Request authentication policies specify the values needed to validate a JSON Web Token (JWT). These values include, among others, the following:

- The location of the token in the request
- The issuer or the request
- The public JSON Web Key Set (JWKS)

Istio checks the presented token, if presented against the rules in the request authentication policy, and rejects requests with invalid tokens. When requests carry no token, they are accepted by default. To

reject requests without tokens, provide authorization rules that specify the restrictions for specific operations, for example paths or actions.

Request authentication policies can specify more than one JWT if each uses a unique location. When more than one policy matches a workload, Istio combines all rules as if they were specified as a single policy. This behavior is useful to program workloads to accept JWT from different providers. However, requests with more than one valid JWT are not supported because the output principal of such requests is undefined.

# Principals

When you use peer authentication policies and mutual TLS, Istio extracts the identity from the peer authentication into the `source.principal`. Similarly, when you use request authentication policies, Istio assigns the identity from the JWT to the `request.auth.principal`. Use these principals to set authorization policies and as telemetry output.

## Updating authentication policies

You can change an authentication policy at any time and Istio pushes the new policies to the workloads almost in real time. However, Istio can't guarantee that all workloads receive the new policy at the same time. The following recommendations help avoid disruption when updating your authentication policies:

- Use intermediate peer authentication policies using the `PERMISSIVE` mode when changing the mode from `DISABLE` to `STRICT` and vice-versa. When all workloads switch successfully to the desired

mode, you can apply the policy with the final mode. You can use Istio telemetry to verify that workloads have switched successfully.

- When migrating request authentication policies from one JWT to another, add the rule for the new JWT to the policy without removing the old rule. Workloads then accept both types of JWT, and you can remove the old rule when all traffic switches to the new JWT. However, each JWT has to use a different location.

# Authorization

Istio's authorization features provide mesh-, namespace-, and workload-wide access control for your workloads in the mesh. This level of control provides the following benefits:

- Workload-to-workload and end-user-to-workload authorization.
- A simple API: it includes a single `AuthorizationPolicy` CRD, which is easy to use and maintain.
- Flexible semantics: operators can define custom

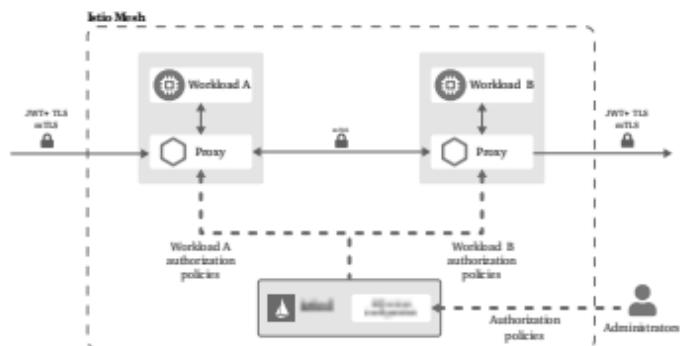
conditions on Istio attributes, and use CUSTOM, DENY and ALLOW actions.

- High performance: Istio authorization (ALLOW and DENY) is enforced natively on Envoy.
- High compatibility: supports gRPC, HTTP, HTTPS and HTTP/2 natively, as well as any plain TCP protocols.

## Authorization architecture

The authorization policy enforces access control to the inbound traffic in the server side Envoy proxy.

Each Envoy proxy runs an authorization engine that authorizes requests at runtime. When a request comes to the proxy, the authorization engine evaluates the request context against the current authorization policies, and returns the authorization result, either ALLOW or DENY. Operators specify Istio authorization policies using .yaml files.



# Authorization Architecture

## **Implicit enablement**

You don't need to explicitly enable Istio's authorization features; they are available after installation. To enforce access control to your workloads, you apply an authorization policy.

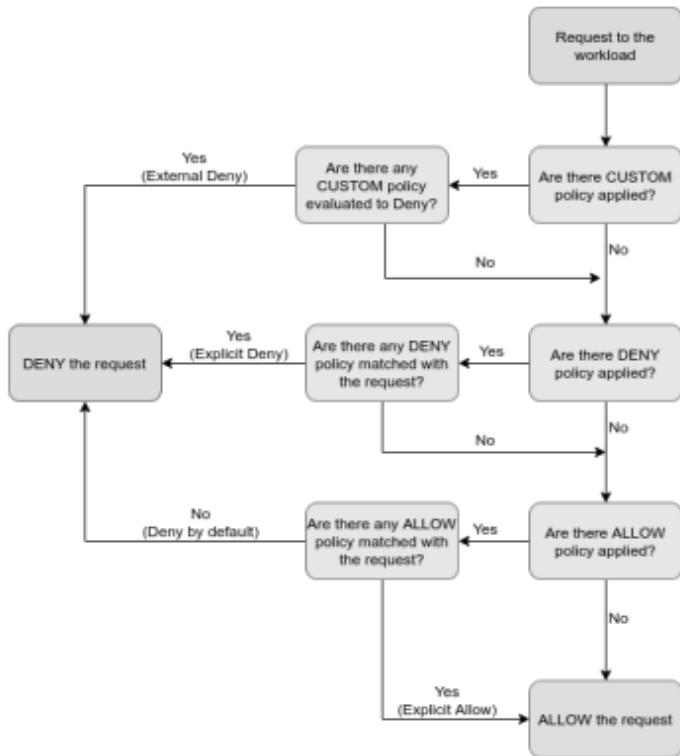
For workloads without authorization policies applied, Istio allows all requests.

Authorization policies support `ALLOW`, `DENY` and `CUSTOM` actions. You can apply multiple policies, each with a different action, as needed to secure access to your workloads.

Istio checks for matching policies in layers, in this order: `CUSTOM`, `DENY`, and then `ALLOW`. For each type of action, Istio first checks if there is a policy with the action applied, and then checks if the request matches the policy's specification. If a request doesn't match a policy in one of the layers, the check continues to the next layer.

The following graph shows the policy precedence in

detail:



## Authorization Policy Precedence

When you apply multiple authorization policies to the same workload, Istio applies them additively.

# Authorization policies

To configure an authorization policy, you create an `AuthorizationPolicy` custom resource. An authorization policy includes a selector, an action, and a list of

rules:

- The `selector` field specifies the target of the policy
- The `action` field specifies whether to allow or deny the request
- The `rules` specify when to trigger the action
  - The `from` field in the `rules` specifies the sources of the request
  - The `to` field in the `rules` specifies the operations of the request
  - The `when` field specifies the conditions needed to apply the rule

The following example shows an authorization policy that allows two sources, the `cluster.local/ns/default/sa/sleep` service account and the `dev` namespace, to access the workloads with the `app: httpbin` and `version: v1` labels in the `foo` namespace when requests sent have a valid JWT token.

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: httpbin
  namespace: foo
spec:
  selector:
    matchLabels:
```

```
app: httpbin
version: v1
action: ALLOW
rules:
- from:
  - source:
    principals: ["cluster.local/ns/default/sa/sleep"]
- source:
  namespaces: ["dev"]
to:
- operation:
  methods: ["GET"]
when:
- key: request.auth.claims[iss]
  values: ["https://accounts.google.com"]
```

The following example shows an authorization policy that denies requests if the source is not the `foo`

## namespace:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: httpbin-deny
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
      version: v1
  action: DENY
  rules:
  - from:
    - source:
        notNamespaces: ["foo"]
```

The deny policy takes precedence over the allow policy. Requests matching allow policies can be denied if they match a deny policy. Istio evaluates deny policies first to ensure that an allow policy can't bypass a deny policy.

## Policy Target

You can specify a policy's scope or target with the metadata/namespace field and an optional selector field. A policy applies to the namespace in the metadata/namespace field. If set its value to the root namespace, the policy applies to all namespaces in a

mesh. The value of the root namespace is configurable, and the default is `istio-system`. If set to any other namespace, the policy only applies to the specified namespace.

You can use a `selector` field to further restrict policies to apply to specific workloads. The `selector` uses labels to select the target workload. The selector contains a list of `{key: value}` pairs, where the `key` is the name of the label. If not set, the authorization policy applies to all workloads in the same namespace as the authorization policy.

For example, the `allow-read` policy allows "GET" and

"HEAD" access to the workload with the app: products label in the default namespace.

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-read
  namespace: default
spec:
  selector:
    matchLabels:
      app: products
  action: ALLOW
  rules:
  - to:
    - operation:
        methods: ["GET", "HEAD"]
```

# Value matching

Most fields in authorization policies support all the following matching schemas:

- Exact match: exact string match.
- Prefix match: a string with an ending "\*". For example, "test.abc.\*" matches "test.abc.com", "test.abc.com.cn", "test.abc.org", etc.
- Suffix match: a string with a starting "\*". For example, "\* .abc .com" matches "eng.abc.com", "test.eng.abc.com", etc.
- Presence match: \* is used to specify anything but not empty. To specify that a field must be present,

use the `fieldname: ["*"]` format. This is different from leaving a field unspecified, which means match anything, including empty.

There are a few exceptions. For example, the following fields only support exact match:

- The `key` field under the `when` section
- The `ipBlocks` under the `source` section
- The `ports` field under the `to` section

The following example policy allows access at paths with the `/test/*` prefix or the `*/info` suffix.

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: tester
  namespace: default
spec:
  selector:
    matchLabels:
      app: products
  action: ALLOW
  rules:
  - to:
    - operation:
        paths: ["/test/*", "*/info"]
```

# Exclusion matching

To match negative conditions like `notValues` in the `when` field, `notIpBlocks` in the `source` field, `notPorts` in the `to` field, Istio supports exclusion matching. The following example requires a valid request principals, which is derived from JWT authentication, if the request path is not `/healthz`. Thus, the policy excludes requests to the `/healthz` path from the JWT authentication:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: disable-jwt-for-healthz
  namespace: default
spec:
  selector:
    matchLabels:
      app: products
  action: ALLOW
  rules:
  - to:
    - operation:
        notPaths: ["/healthz"]
  from:
  - source:
      requestPrincipals: ["*"]
```

The following example denies the request to the

/admin path for requests without request principals:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: enable-jwt-for-admin
  namespace: default
spec:
  selector:
    matchLabels:
      app: products
  action: DENY
  rules:
  - to:
    - operation:
      paths: ["/admin"]
  from:
  - source:
    notRequestPrincipals: ["*"]
```

## allow-nothing, deny-all and allow-all policy

The following example shows an `ALLOW` policy that matches nothing. If there are no other `ALLOW` policies, requests will always be denied because of the “deny by default” behavior.

Note the “deny by default” behavior applies only if the workload has at least one authorization policy with the `ALLOW` action.

It is a good security practice to start with the allow-nothing policy and incrementally add more `ALLOW` policies to open more access to the workload.

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-nothing
spec:
  action: ALLOW
  # the rules field is not specified, and the policy will never
  # match.
```

The following example shows a `DENY` policy that explicitly denies all access. It will always deny the request even if there is another `ALLOW` policy allowing the request because the `DENY` policy takes precedence over the `ALLOW` policy. This is useful if you want to temporarily disable all access to the workload.

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: deny-all
spec:
  action: DENY
  # the rules field has an empty rule, and the policy will always match.
  rules:
  - {}
```

The following example shows an `ALLOW` policy that allows full access to the workload. It will make other `ALLOW` policies useless as it will always allow the request. It might be useful if you want to temporarily expose full access to the workload. Note the request

could still be denied due to `CUSTOM` and `DENY` policies.

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-all
spec:
  action: ALLOW
  # This matches everything.
  rules:
  - {}
```

## Custom conditions

You can also use the `when` section to specify additional conditions. For example, the following

AuthorizationPolicy definition includes a condition that request.headers[version] is either "v1" or "v2". In this case, the key is request.headers[version], which is an entry in the Istio attribute request.headers, which is a map.

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: httpbin
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
      version: v1
  action: ALLOW
  rules:
```

```
- from:  
  - source:  
    principals: ["cluster.local/ns/default/sa/sleep"]  
  to:  
  - operation:  
    methods: ["GET"]  
when:  
- key: request.headers[version]  
  values: ["v1", "v2"]
```

The supported key values of a condition are listed on the conditions page.

# Authenticated and unauthenticated identity

If you want to make a workload publicly accessible, you need to leave the `source` section empty. This allows sources from all (both authenticated and unauthenticated) users and workloads, for example:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: httpbin
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
      version: v1
  action: ALLOW
  rules:
  - to:
    - operation:
        methods: ["GET", "POST"]
```

To allow only authenticated users, set principals to "\*" instead, for example:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: httpbin
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
      version: v1
  action: ALLOW
  rules:
    - from:
        - source:
            principals: ["*"]
        to:
          - operation:
              methods: ["GET", "POST"]
```

# Using Istio authorization on plain TCP protocols

Istio authorization supports workloads using any plain TCP protocols, such as MongoDB. In this case, you configure the authorization policy in the same way you did for the HTTP workloads. The difference is that certain fields and conditions are only applicable to HTTP workloads. These fields include:

- The `request_principals` field in the source section of the authorization policy object
- The `hosts`, `methods` and `paths` fields in the operation

## section of the authorization policy object

The supported conditions are listed in the [conditions](#) page. If you use any HTTP only fields for a TCP workload, Istio will ignore HTTP-only fields in the authorization policy.

Assuming you have a MongoDB service on port `27017`, the following example configures an authorization policy to only allows the `bookinfo-ratings-v2` service in the Istio mesh to access the MongoDB workload.

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: mongodb-policy
  namespace: default
spec:
  selector:
    matchLabels:
      app: mongodb
  action: ALLOW
  rules:
  - from:
      - source:
          principals: ["cluster.local/ns/default/sa/bookinfo-rating
s-v2"]
        to:
        - operation:
            ports: ["27017"]
```

# Dependency on mutual TLS

Istio uses mutual TLS to securely pass some information from the client to the server. Mutual TLS must be enabled before using any of the following fields in the authorization policy:

- the `principals` and `notPrincipals` field under the source **section**
- the `namespaces` and `notNamespaces` field under the source **section**
- the `source.principal` custom condition
- the `source.namespace` custom condition

Note it is strongly recommended to always use these fields with **strict** mutual TLS mode in the PeerAuthentication to avoid potential unexpected requests rejection or policy bypass when plain text traffic is used with the permissive mutual TLS mode.

Check the security advisory for more details and alternatives if you cannot enable strict mutual TLS mode.

## Learn more

After learning the basic concepts, there are more resources to review:

- Try out the security policy by following the authentication **and** authorization tasks.
- Learn some security policy examples **that could be** used to improve security in your mesh.
- Read common problems to better troubleshoot security policy issues when something goes wrong.



# Observability

⌚ 5 minute read

---

Istio generates detailed telemetry for all service communications within a mesh. This telemetry provides *observability* of service behavior, empowering operators to troubleshoot, maintain, and optimize their applications - without imposing any additional burdens on service developers. Through Istio, operators gain a thorough understanding of how

monitored services are interacting, both with other services and with the Istio components themselves.

Istio generates the following types of telemetry in order to provide overall service mesh observability:

- **Metrics.** Istio generates a set of service metrics based on the four “golden signals” of monitoring (latency, traffic, errors, and saturation). Istio also provides detailed metrics for the mesh control plane. A default set of mesh monitoring dashboards built on top of these metrics is also provided.
- **Distributed Traces.** Istio generates distributed trace spans for each service, providing operators with a

detailed understanding of call flows and service dependencies within a mesh.

- **Access Logs.** As traffic flows into a service within a mesh, Istio can generate a full record of each request, including source and destination metadata. This information enables operators to audit service behavior down to the individual workload instance level.

## Metrics

Metrics provide a way of monitoring and understanding behavior in aggregate.

To monitor service behavior, Istio generates metrics for all service traffic in, out, and within an Istio service mesh. These metrics provide information on behaviors such as the overall volume of traffic, the error rates within the traffic, and the response times for requests.

In addition to monitoring the behavior of services within a mesh, it is also important to monitor the behavior of the mesh itself. Istio components export metrics on their own internal behaviors to provide

insight on the health and function of the mesh control plane.

## Proxy-level metrics

Istio metrics collection begins with the sidecar proxies (Envoy). Each proxy generates a rich set of metrics about all traffic passing through the proxy (both inbound and outbound). The proxies also provide detailed statistics about the administrative functions of the proxy itself, including configuration and health information.

Envoy-generated metrics provide monitoring of the mesh at the granularity of Envoy resources (such as listeners and clusters). As a result, understanding the connection between mesh services and Envoy resources is required for monitoring the Envoy metrics.

Istio enables operators to select which of the Envoy metrics are generated and collected at each workload instance. By default, Istio enables only a small subset of the Envoy-generated statistics to avoid overwhelming metrics backends and to reduce the CPU overhead associated with metrics collection. However, operators can easily expand the set of

collected proxy metrics when required. This enables targeted debugging of networking behavior, while reducing the overall cost of monitoring across the mesh.

The Envoy documentation site includes a detailed overview of Envoy statistics collection. The operations guide on Envoy Statistics provides more information on controlling the generation of proxy-level metrics.

Example proxy-level Metrics:

```
envoy_cluster_internal_upstream_rq{response_code_class="2xx", cluster_name="xds-grpc"} 7163  
  
envoy_cluster_upstream_rq_completed{cluster_name="xds-grpc"} 7164  
  
envoy_cluster_ssl_connection_error{cluster_name="xds-grpc"} 0  
  
envoy_cluster_lb_subsets_removed{cluster_name="xds-grpc"} 0  
  
envoy_cluster_internal_upstream_rq{response_code="503", cluster_name="xds-grpc"} 1
```

# Service-level metrics

In addition to the proxy-level metrics, Istio provides a set of service-oriented metrics for monitoring service communications. These metrics cover the four basic service monitoring needs: latency, traffic, errors, and saturation. Istio ships with a default set of dashboards for monitoring service behaviors based on these metrics.

The standard Istio metrics are exported to Prometheus by default.

Use of the service-level metrics is entirely optional. Operators may choose to turn off generation and collection of these metrics to meet their individual

needs.

## Example service-level metric:

```
istio_requests_total{  
    connection_security_policy="mutual_tls",  
    destination_app="details",  
    destination_canonical_service="details",  
    destination_canonical_revision="v1",  
    destination_principal="cluster.local/ns/default/sa/default",  
    destination_service="details.default.svc.cluster.local",  
    destination_service_name="details",  
    destination_service_namespace="default",  
    destination_version="v1",  
    destination_workload="details-v1",  
    destination_workload_namespace="default",  
    reporter="destination",  
    request_protocol="http",  
    response_code="200",
```

```
response_flags="-",
source_app="productpage",
source_canonical_service="productpage",
source_canonical_revision="v1",
source_principal="cluster.local/ns/default/sa/default",
source_version="v1",
source_workload="productpage-v1",
source_workload_namespace="default"
} 214
```

## Control plane metrics

The Istio control plane also provides a collection of self-monitoring metrics. These metrics allow monitoring of the behavior of Istio itself (as distinct

from that of the services within the mesh).

For more information on which metrics are maintained, please refer to the reference documentation.

## Distributed traces

Distributed tracing provides a way to monitor and understand behavior by monitoring individual requests as they flow through a mesh. Traces empower mesh operators to understand service

dependencies and the sources of latency within their service mesh.

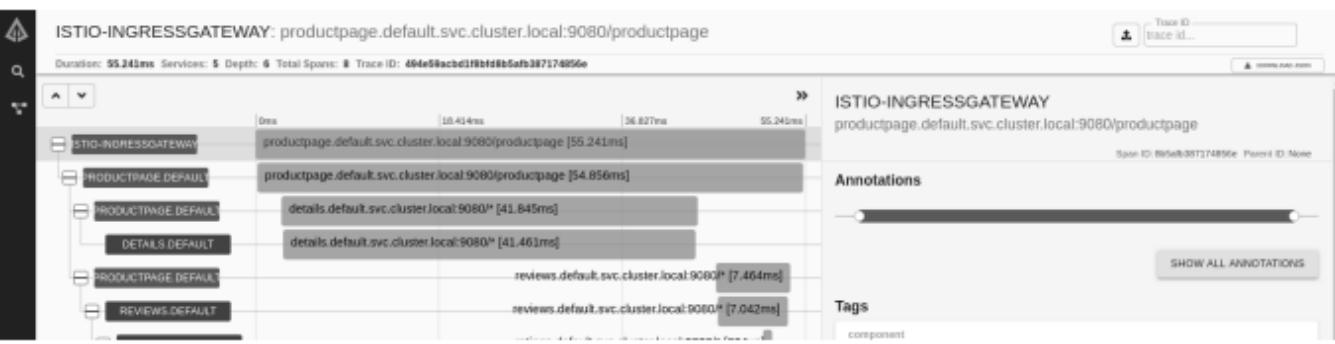
Istio supports distributed tracing through the Envoy proxies. The proxies automatically generate trace spans on behalf of the applications they proxy, requiring only that the applications forward the appropriate request context.

Istio supports a number of tracing backends, including Zipkin, Jaeger, Lightstep, and Datadog. Operators control the sampling rate for trace generation (that is, the rate at which tracing data is generated per request). This allows operators to

control the amount and rate of tracing data being produced for their mesh.

More information about Distributed Tracing with Istio is found in our FAQ on Distributed Tracing.

Example Istio-generated distributed trace for a single request:





Distributed Trace for a single request

# Access logs

Access logs provide a way to monitor and understand

behavior from the perspective of an individual workload instance.

Istio can generate access logs for service traffic in a configurable set of formats, providing operators with full control of the how, what, when and where of logging. For more information, please refer to [Getting Envoy's Access Logs](#).

Example Istio access log:

[2019-03-06T09:31:27.360Z] "GET /status/418 HTTP/1.1" 418 - "-"  
0 135 5 2 "-" "curl/7.60.0" "d209e46f-9ed5-9b61-bbdd-43e22662702  
a" "httpbin:8000" "127.0.0.1:80" inbound|8000|http|httpbin.defau  
lt.svc.cluster.local - 172.30.146.73:80 172.30.146.82:38618 outb  
ound\_.8000\_.\_.httpbin.default.svc.cluster.local

# Extensibility

⌚ 2 minute read

---

WebAssembly is a sandboxing technology which can be used to extend the Istio proxy (Envoy). The Proxy-Wasm sandbox API replaces Mixer as the primary extension mechanism in Istio.

WebAssembly sandbox goals:

- **Efficiency** - An extension adds low latency, CPU,

and memory overhead.

- **Function** - An extension can enforce policy, collect telemetry, and perform payload mutations.
- **Isolation** - A programming error or crash in one plugin doesn't affect other plugins.
- **Configuration** - The plugins are configured using an API that is consistent with other Istio APIs. An extension can be configured dynamically.
- **Operator** - An extension can be canaried and deployed as log-only, fail-open or fail-close.
- **Extension developer** - The plugin can be written in several programming languages.

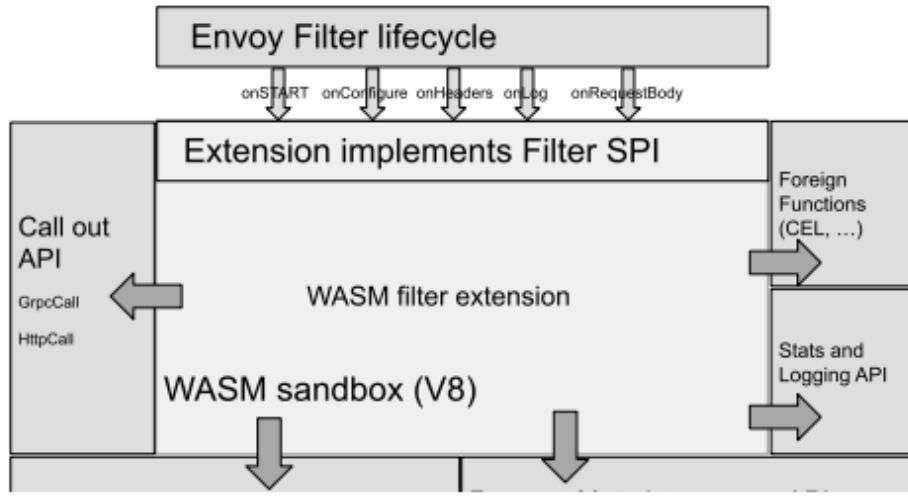
This video talk is an introduction about architecture of WebAssembly integration.

# High-level architecture

Istio extensions (Proxy-Wasm plugins) have several components:

- **Filter Service Provider Interface (SPI)** for building Proxy-Wasm plugins for filters.
- **Sandbox** V8 Wasm Runtime embedded in Envoy.

- **Host APIs** for headers, trailers and metadata.
- **Call out APIs** for gRPC and HTTP calls.
- **Stats and Logging APIs** for metrics and monitoring.



Header and Trailer access API  
getRequestHeaders, setRequestHeaders...

Request Metadata access API  
getRequestMetadata, setRequestMetadata...

## Extending Istio/Envoy

# Example

An example C++ Proxy-Wasm plugin for a filter can be found [here](#). You can follow this guide [to implement a Wasm extension with C++](#).

# Ecosystem

- Istio Ecosystem Wasm Extensions
- Proxy-Wasm ABI specification
- Proxy-Wasm C++ SDK
- Proxy-Wasm Rust SDK
- Proxy-Wasm AssemblyScript SDK
- WebAssembly Hub
- WebAssembly Extensions For Network Proxies (video)

# Getting Started

⌚ 8 minute read ✓ page test

---

This guide lets you quickly evaluate Istio. If you are already familiar with Istio or interested in installing other configuration profiles or advanced deployment models, refer to our [which Istio installation method should I use? FAQ page](#).

These steps require you to have a cluster running a

supported version of Kubernetes (1.19, 1.20, 1.21, 1.22). You can use any supported platform, for example Minikube or others specified by the platform-specific setup instructions.

Follow these steps to get started with Istio:

1. Download and install Istio
2. Deploy the sample application
3. Open the application to outside traffic
4. View the dashboard

# Download Istio

1. Go to the Istio release page to download the installation file for your OS, or download and extract the latest release automatically (Linux or macOS):

```
$ curl -L https://istio.io/downloadIstio | sh -
```

The command above downloads the latest release (numerically) of Istio. You can pass variables on the command line



to download a specific version or to override the processor architecture. For example, to download Istio 1.6.8 for the `x86_64` architecture, run:

```
$ curl -L https://istio.io/downloadIstio | ISTIO_VERSION=1.6.8 TARGET_ARCH=x86_64 sh -
```

2. Move to the Istio package directory. For example, if the package is `istio-1.11.3`:

```
$ cd istio-1.11.3
```

The installation directory contains:

- Sample applications in `samples/`
- The `istioctl` client binary in the `bin/` directory.

3. Add the `istioctl` client to your path (Linux or macOS):

```
$ export PATH=$PWD/bin:$PATH
```

## Install Istio

1. For this installation, we use the `demo` configuration

profile. It's selected to have a good set of defaults for testing, but there are other profiles for production or performance testing.



If your platform has a vendor-specific configuration profile, e.g., Openshift, use it in the following command, instead of the `demo` profile. Refer to your platform instructions for details.

```
$ istioctl install --set profile=demo -y
✓ Istio core installed
✓ Istiod installed
✓ Egress gateways installed
✓ Ingress gateways installed
✓ Installation complete
```

2. Add a namespace label to instruct Istio to automatically inject Envoy sidecar proxies when you deploy your application later:

```
$ kubectl label namespace default istio-injection=enabled
namespace/default labeled
```

# Deploy the sample application

1. Deploy the Bookinfo sample application:

```
$ kubectl apply -f @samples/bookinfo/platform/kube/bookinfo
.yaml@  
service/details created  
serviceaccount/bookinfo-details created  
deployment.apps/details-v1 created  
service/ratings created  
serviceaccount/bookinfo-ratings created  
deployment.apps/ratings-v1 created  
service/reviews created  
serviceaccount/bookinfo-reviews created  
deployment.apps/reviews-v1 created  
deployment.apps/reviews-v2 created  
deployment.apps/reviews-v3 created  
service/productpage created  
serviceaccount/bookinfo-productpage created  
deployment.apps/productpage-v1 created
```

2. The application will start. As each pod becomes ready, the Istio sidecar will be deployed along

with it.

```
$ kubectl get services
```

NAME T(S)	TYPE AGE	CLUSTER-IP	EXTERNAL-IP	PORTS
details	ClusterIP 0/TCP 29s	10.0.0.212	<none>	9080
kubernetes	ClusterIP /TCP 25m	10.0.0.1	<none>	443
productpage	ClusterIP 0/TCP 28s	10.0.0.57	<none>	9080
ratings	ClusterIP 0/TCP 29s	10.0.0.33	<none>	9080
reviews	ClusterIP 0/TCP 29s	10.0.0.28	<none>	9080

and

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS
S AGE details-v1-558b8b4b76-2111d 2m41s	2/2	Running	0
productpage-v1-6987489c74-1pkgl 2m40s	2/2	Running	0
ratings-v1-7dc98c7588-vzftc 2m41s	2/2	Running	0
reviews-v1-7f99cc4496-gdxfn 2m41s	2/2	Running	0
reviews-v2-7d79d5bd5d-8zzqd 2m41s	2/2	Running	0
reviews-v3-7dbcdcbc56-m8dph 2m41s	2/2	Running	0

Re-run the previous command and wait



until all pods report READY 2/2 and STATUS Running before you go to the next step. This might take a few minutes depending on your platform.

3. Verify everything is working correctly up to this point. Run this command to see if the app is running inside the cluster and serving HTML pages by checking for the page title in the response:

```
$ kubectl exec "$(kubectl get pod -l app=ratings -o jsonpath='{.items[0].metadata.name}')" -c ratings -- curl -sS productpage:9080/productpage | grep -o "<title>.*</title>"  
<title>Simple Bookstore App</title>
```

## Open the application to outside traffic

The Bookinfo application is deployed but not accessible from the outside. To make it accessible, you need to create an Istio Ingress Gateway, which maps a path to a route at the edge of your mesh.

## 1. Associate this application with the Istio gateway:

```
$ kubectl apply -f @samples/bookinfo/networking/bookinfo-gateway.yaml@  
gateway.networking.istio.io/bookinfo-gateway created  
virtualservice.networking.istio.io/bookinfo created
```

## 2. Ensure that there are no issues with the configuration:

```
$ istioctl analyze  
✓ No validation issues found when analyzing namespace: default.
```

# Determining the ingress IP

# and ports

Follow these instructions to set the `INGRESS_HOST` and `INGRESS_PORT` variables for accessing the gateway. Use the tabs to choose the instructions for your chosen platform:

**Minikube**

Other platforms

Set the ingress ports:

```
$ export INGRESS_PORT=$(kubectl -n istio-system get service istio-ingressgateway -o jsonpath='{.spec.ports[?(@.name=="http2")].nodePort}')
$ export SECURE_INGRESS_PORT=$(kubectl -n istio-system get service istio-ingressgateway -o jsonpath='{.spec.ports[?(@.name=="https")].nodePort}')
```

Ensure a port was successfully assigned to each environment variable:

```
$ echo "$INGRESS_PORT"
32194
```

```
$ echo "$SECURE_INGRESS_PORT"
31632
```

Set the ingress IP:

```
$ export INGRESS_HOST=$(minikube ip)
```

Ensure an IP address was successfully assigned to the environment variable:

```
$ echo "$INGRESS_HOST"  
192.168.4.102
```

Run this command in a new terminal window to start a Minikube tunnel that sends traffic to your Istio Ingress Gateway:

```
$ minikube tunnel
```

## 1. Set GATEWAY\_URL:

```
$ export GATEWAY_URL=$INGRESS_HOST:$INGRESS_PORT
```

## 2. Ensure an IP address and port were successfully assigned to the environment variable:

```
$ echo "$GATEWAY_URL"  
192.168.99.100:32194
```

# Verify external access

Confirm that the Bookinfo application is accessible from outside by viewing the Bookinfo product page using a browser.

1. Run the following command to retrieve the external address of the Bookinfo application.

```
$ echo "http://$GATEWAY_URL/productpage"
```

2. Paste the output from the previous command into your web browser and confirm that the Bookinfo product page is displayed.

# View the dashboard

Istio integrates with several different telemetry applications. These can help you gain an understanding of the structure of your service mesh, display the topology of the mesh, and analyze the health of your mesh.

Use the following instructions to deploy the Kiali dashboard, along with Prometheus, Grafana, and Jaeger.

1. Install Kiali and the other addons and wait for them to be deployed.

```
$ kubectl apply -f samples/addons
$ kubectl rollout status deployment/kiali -n istio-system
Waiting for deployment "kiali" rollout to finish: 0 of 1 up
dated replicas are available...
deployment "kiali" successfully rolled out
```

If there are errors trying to install the addons, try running the command again.

- (i) There may be some timing issues which will be resolved when the command is run again.

2. Access the Kiali dashboard.

```
$ istioctl dashboard kiali
```

3. In the left navigation menu, select *Graph* and in the *Namespace* drop down, select *default*.

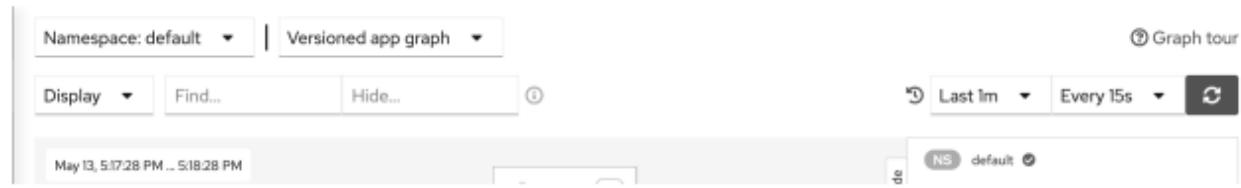
To see trace data, you must send requests to your service. The number of requests depends on Istio's sampling rate. You set this rate when you install Istio. The default sampling rate is 1%.

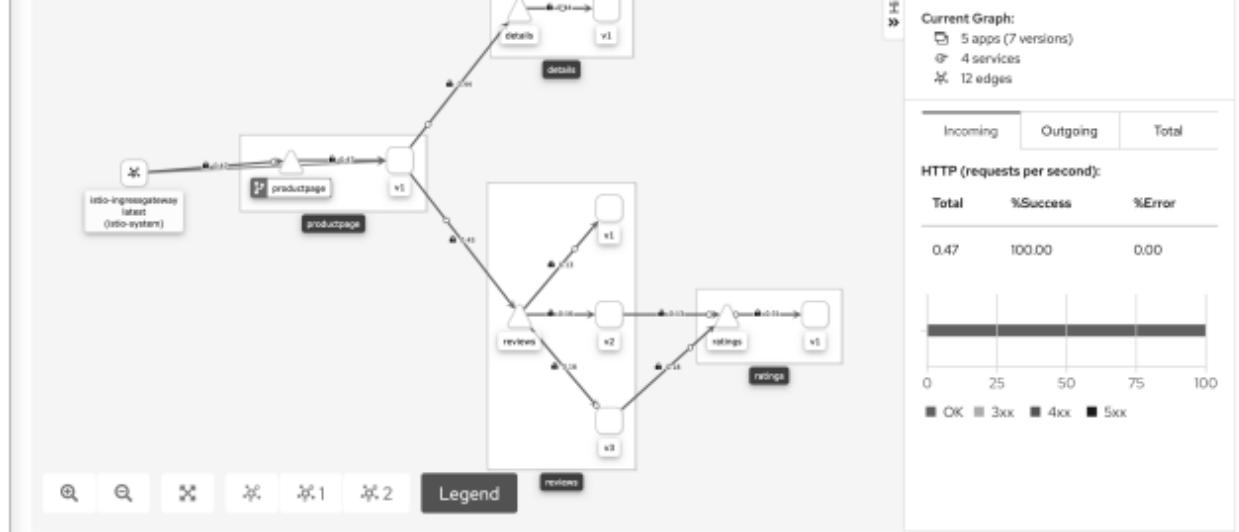
 You need to send at least 100 requests before the first trace is visible. To send a 100 requests to the productpage service,

use the following command:

```
$ for i in $(seq 1 100); do curl -s -o /dev/null  
"http://$GATEWAY_URL/productpage"; done
```

The Kiali dashboard shows an overview of your mesh with the relationships between the services in the Bookinfo sample application. It also provides filters to visualize the traffic flow.





## Kiali Dashboard

# Next steps

Congratulations on completing the evaluation installation!

These tasks are a great place for beginners to further evaluate Istio's features using this `demo` installation:

- Request routing
- Fault injection
- Traffic shifting
- Querying metrics
- Visualizing metrics
- Accessing external services
- Visualizing your mesh

Before you customize Istio for production use, see these resources:

- Deployment models
- Deployment best practices
- Pod requirements
- General installation instructions

**Join the Istio community**

We welcome you to ask questions and give us feedback by joining the Istio community.

## Uninstall

To delete the Bookinfo sample application and its configuration, see Bookinfo cleanup.

The Istio uninstall deletes the RBAC permissions and all resources hierarchically under the `istio-system` namespace. It is safe to ignore errors for non-existent

resources because they may have been deleted hierarchically.

```
$ kubectl delete -f @samples/addons@  
$ istioctl manifest generate --set profile=demo | kubectl delete  
--ignore-not-found=true -f -
```

The `istio-system` namespace is not removed by default. If no longer needed, use the following command to remove it:

```
$ kubectl delete namespace istio-system
```

The label to instruct Istio to automatically inject Envoy sidecar proxies is not removed by default. If no

longer needed, use the following command to remove it:

```
$ kubectl label namespace default istio-injection-
```

# kind

⌚ 3 minute read  page test

---

kind is a tool for running local Kubernetes clusters using Docker container nodes. kind was primarily designed for testing Kubernetes itself, but may be used for local development or CI. Follow these instructions to prepare a kind cluster for Istio installation.

# Prerequisites

- Please use the latest Go version.
- To use kind, you will also need to install docker.
- Install the latest version of kind.
- Increase Docker's memory limit.

# Installation steps

1. Create a cluster with the following command:

```
$ kind create cluster --name istio-testing
```

--name is used to assign a specific name to the cluster. By default, the cluster will be given the name kind.

2. To see the list of kind clusters, use the following command:

```
$ kind get clusters  
istio-testing
```

3. To list the local Kubernetes contexts, use the following command.

```
$ kubectl config get-contexts
```

CURRENT	NAME	CLUSTER	AUTHINFO
0	NAMESPACE		
*	kind-istio-testing	kind-istio-testing	kind-istio-testing
	minikube	minikube	minikube
e			

 kind is prefixed to the context and cluster names, for example: kind-istio-testing

4. If you run multiple clusters, you need to choose which cluster `kubectl` talks to. You can set a default cluster for `kubectl` by setting the current

context in the Kubernetes kubeconfig file.

Additionally you can run following command to set the current context for kubectl.

```
$ kubectl config use-context kind-istio-testing  
Switched to context "kind-istio-testing".
```

Once you are done setting up a kind cluster, you can proceed to install Istio on it.

5. When you are done experimenting and you want to delete the existing cluster, use the following command:

```
$ kind delete cluster --name istio-testing  
Deleting cluster "istio-testing" ...
```

# Setup Dashboard UI for kind

kind does not have a built in Dashboard UI like minikube. But you can still setup Dashboard, a web based Kubernetes UI, to view your cluster. Follow these instructions to setup Dashboard for kind.

1. To deploy Dashboard, run the following command:

```
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.1.0/aio/deploy/recommended.yaml
```

## 2. Verify that Dashboard is deployed and running.

```
$ kubectl get pod -n kubernetes-dashboard
NAME                                         READY   STATUS
RESTARTS   AGE
dashboard-metrics-scraper-76585494d8-zdb66   1/1    Running
g          0        39s
kubernetes-dashboard-b7ffbc8cb-zl8zg         1/1    Running
g          0        39s
```

## 3. Create a ClusterRoleBinding to provide admin access to the newly created cluster.

```
$ kubectl create clusterrolebinding default-admin --cluster
role cluster-admin --serviceaccount=default:default
```

## 4. To login to Dashboard, you need a Bearer Token.

Use the following command to store the token in a variable.

```
$ token=$(kubectl get secrets -o jsonpath=".items[?(@.meta  
data.annotations['kubernetes\\.io/service-account\\.name']=='  
default')].data.token"}|base64 --decode)
```

Display the token using the echo command and copy it to use for logging into Dashboard.

```
$ echo $token
```

5. You can Access Dashboard using the kubectl command-line tool by running the following command:

```
$ kubectl proxy
```

```
Starting to serve on 127.0.0.1:8001
```

Click Kubernetes Dashboard [to view your deployments and services.](#)



You have to save your token somewhere, otherwise you have to run step number 4 everytime you need a token to login to your Dashboard.



# Install with Helm

⌚ 5 minute read ✓ page test

---

Follow this guide to install and configure an Istio mesh using Helm for in-depth evaluation.

The Helm charts used in this guide are the same underlying charts used when installing Istio via `Istioctl` or the Operator.

This feature is currently considered alpha.



Prior to Istio 1.9.0, installations using the Helm charts required hub and tag arguments: `--set global.hub="docker.io/istio"` and `--set global.tag="1.8.2"`. As of Istio 1.9.0 these are no longer required.

## Prerequisites

1. Download the Istio release.
2. Perform any necessary platform-specific setup.
3. Check the Requirements for Pods and Services.
4. Install a Helm client **with a version higher than 3.1.1**.

 Helm 2 is not supported for installing Istio.  
The commands in this guide use the Helm charts that are included in the Istio release package located at manifests/charts.

# Installation steps

Change directory to the root of the release package and then follow the instructions below.

The default chart configuration uses the secure third party tokens for the service account token projections used by Istio proxies to authenticate with the Istio control plane. Before proceeding to install any of the charts below, you should verify if third party tokens are enabled in your cluster by

following the steps described here. If third party tokens are not enabled, you should add the option `--set global.jwtPolicy=first-party-jwt` to the Helm install commands. If the `jwtPolicy` is not set correctly, pods associated with `istiod`, gateways or workloads with injected Envoy proxies will not get deployed due to the missing `istio-token` volume.

1. Create a namespace `istio-system` for Istio components:

```
$ kubectl create namespace istio-system
```

2. Install the Istio base chart which contains cluster-wide resources used by the Istio control plane:

```
$ helm install istio-base manifests/charts/base -n istio-system
```

3. Install the Istio discovery chart which deploys the `istiod` service:

```
$ helm install istiod manifests/charts/istio-control/istio-discovery \
-n istio-system
```

4. (Optional) Install the Istio ingress gateway chart which contains the ingress gateway components:

```
$ helm install istio-ingress manifests/charts/gateways/istio-ingress \
    -n istio-system
```

5. (Optional) Install the Istio egress gateway chart which contains the egress gateway components:

```
$ helm install istio-egress manifests/charts/gateways/istio-egress \
    -n istio-system
```

## Verifying the installation

Ensure all Kubernetes pods in `istio-system` namespace are deployed and have a STATUS of Running:

```
$ kubectl get pods -n istio-system
```

## Updating your Istio configuration

You can provide override settings specific to any Istio Helm chart used above and follow the Helm upgrade workflow to customize your Istio mesh installation.

The available configurable options can be found by inspecting the top level `values.yaml` file associated with the Helm charts located at `manifests/charts` inside the Istio release package specific to your version.



Note that the Istio Helm chart values are under active development and considered experimental. Upgrading to newer versions of Istio can involve migrating your override values to follow the new API.

For customizations that are supported via both `ProxyConfig` and Helm values, using `ProxyConfig` is recommended because it provides schema validation while unstructured Helm values do not.

## Create a backup

Before upgrading Istio in your cluster, we recommend creating a backup of your custom configurations, and restoring it from backup if necessary:

```
$ kubectl get istio-io --all-namespaces -oyaml > "$HOME"/istio_resource_backup.yaml
```

You can restore your custom configuration like this:

```
$ kubectl apply -f "$HOME"/istio_resource_backup.yaml
```

## Migrating from non-Helm installations

If you're migrating from a version of Istio installed using `istioctl` or Operator to Helm (Istio 1.5 or

earlier), you need to delete your current Istio control plane resources and re-install Istio using Helm as described above. When deleting your current Istio installation, you must not remove the Istio Custom Resource Definitions (CRDs) as that can lead to loss of your custom Istio resources.



It is highly recommended to take a backup of your Istio resources using steps described above before deleting current Istio installation in your cluster.

You can follow steps mentioned in the `Istioctl uninstall` guide or Operator uninstall guide depending upon your installation method.

## Uninstall

You can uninstall Istio and its components by uninstalling the charts installed above.

1. List all the Istio charts installed in `istio-system` namespace:

```
$ helm ls -n istio-system
```

NAME	NAMESPACE	REVISION	UPDATED
APP VERSION		STATUS	CHART
istio-base	istio-system	1	...
		deployed	base-1.9.0
istio-egress	istio-system	1	...
		deployed	istio-egress-1.9.0
istio-ingress	istio-system	1	...
		deployed	istio-ingress-1.9.0
istiod	istio-system	1	...
		deployed	istio-discovery-1.9.0

## 2. (Optional) Delete Istio ingress/egress chart:

```
$ helm delete istio-egress -n istio-system  
$ helm delete istio-ingress -n istio-system
```

### 3. Delete Istio discovery chart:

```
$ helm delete istiod -n istio-system
```

### 4. Delete Istio base chart:

 By design, deleting a chart via Helm doesn't delete the installed Custom Resource Definitions (CRDs) installed via the chart.

```
$ helm delete istio-base -n istio-system
```

## 5. Delete the `istio-system` namespace:

```
$ kubectl delete namespace istio-system
```

# Uninstall stable revision label resources

If you decide to continue using the old control plane, instead of completing the update, you can uninstall the newer revision and its tag by first issuing `helm template istiod manifests/charts/istio-control/istio-`

```
discovery -s templates/revision-tags.yaml --set  
revisionTags={prod-canary} --set revision=canary -n  
istio-system | kubectl delete -f -. You must then  
uninstall the revision of Istio that it pointed to by  
following the uninstall procedure above.
```

If you installed the gateway(s) for this revision using in-place upgrades, you must also reinstall the gateway(s) for the previous revision manually. Removing the previous revision and its tags will not automatically revert the previously in-place upgraded gateway(s).

# (Optional) Deleting CRDs installed by Istio

Deleting CRDs permanently removes any Istio resources you have created in your cluster. To permanently delete Istio CRDs installed in your cluster:

```
$ kubectl get crd | grep --color=never 'istio.io' | awk '{print $1}' \
| xargs -n1 kubectl delete crd
```

# Canary Upgrades

⌚ 6 minute read  page test

---

Upgrading Istio can be done by first running a canary deployment of the new control plane, allowing you to monitor the effect of the upgrade with a small percentage of the workloads before migrating all of the traffic to the new version. This is much safer than doing an in-place upgrade and is the recommended upgrade method.

When installing Istio, the `revision` installation setting can be used to deploy multiple independent control planes at the same time. A canary version of an upgrade can be started by installing the new Istio version's control plane next to the old one, using a different `revision` setting. Each revision is a full Istio control plane implementation with its own `Deployment`, `Service`, etc.

## Before you upgrade

Before upgrading Istio, it is recommended to run the `istioctl x precheck` command to make sure the upgrade is compatible with your environment.

```
$ istioctl x precheck
✓ No issues found when checking the cluster. Istio is safe to install or upgrade!
To get started, check out https://istio.io/latest/docs/setup/getting-started/
```

When using revision-based upgrades jumping across two patch versions is supported (e.g. upgrading directly from version 1.8 to 1.10). This is in contrast to in-

place upgrades where it is required to upgrade to each intermediate patch release.

## Control plane

To install a new revision called canary, you would set the `revision` field as follows:

In a production environment, a better

i revision name would correspond to the Istio version. However, you must replace . characters in the revision name, for example, revision=1-6-8 for Istio 1.6.8, because . is not a valid revision name character.

```
$ istioctl install --set revision=canary
```

After running the command, you will have two control plane deployments and services running side-by-side:

```
$ kubectl get pods -n istio-system -l app=istiod
NAME                               READY   STATUS    RESTARTS
TS      AGE
istiod-786779888b-p9s5n          1/1     Running   0
    114m
istiod-canary-6956db645c-vwhsk  1/1     Running   0
    1m
```

```
$ kubectl get svc -n istio-system -l app=istiod
NAME        TYPE        CLUSTER-IP      EXTERNAL-IP      PORT(S)
          AGE
istiod      ClusterIP   10.32.5.247  <none>          15010/TCP
P, 15012/TCP, 443/TCP, 15014/TCP           33d
istiod-canary  ClusterIP   10.32.6.58   <none>          15010/TCP
P, 15012/TCP, 443/TCP, 15014/TCP, 53/UDP, 853/TCP  12m
```

You will also see that there are two sidecar injector configurations including the new revision.

```
$ kubectl get mutatingwebhookconfigurations
```

NAME	WEBHOOKS	AGE
istio-sidecar-injector	1	7m56s
istio-sidecar-injector-canary	1	3m18s

## Data plane

Unlike `istiod`, Istio gateways do not run revision-specific instances, but are instead in-place upgraded to use the new control plane revision. You can verify that the `istio-ingress` gateway is using the `canary` revision by running the following command:

```
$ istioctl proxy-status | grep $(kubectl -n istio-system get pod  
-l app=istio-ingressgateway -o jsonpath='{.items..metadata.name  
'}) | awk '{print $7}'  
istiod-canary-6956db645c-vwhsk
```

However, simply installing the new revision has no impact on the existing sidecar proxies. To upgrade these, you must configure them to point to the new `istiod-canary` control plane. This is controlled during sidecar injection based on the namespace label `istio.io/rev`.

To upgrade the namespace `test-ns`, remove the `istio-injection` label, and add the `istio.io/rev` label to point to the canary revision. The `istio-injection` label must

be removed because it takes precedence over the `istio.io/rev` label for backward compatibility.

```
$ kubectl label namespace test-ns istio-injection- istio.io/rev=canary
```

After the namespace updates, you need to restart the pods to trigger re-injection. One way to do this is using:

```
$ kubectl rollout restart deployment -n test-ns
```

When the pods are re-injected, they will be configured to point to the `istiod-canary` control plane.

You can verify this by looking at the pod labels.

For example, the following command will show all the pods using the `canary` revision:

```
$ kubectl get pods -n test-ns -l istio.io/rev=canary
```

To verify that the new pods in the `test-ns` namespace are using the `istiod-canary` service corresponding to the `canary` revision, select one newly created pod and use the `pod_name` in the following command:

```
$ istioctl proxy-status | grep ${pod_name} | awk '{print $7}'  
istiod-canary-6956db645c-vwhsk
```

The output confirms that the pod is using `istiod-canary` revision of the control plane.

## **Stable revision labels (experimental)**

- ⓘ If you're using Helm, refer to the Helm upgrade documentation.

Manually relabeling namespaces when moving them to a new revision can be tedious and error-prone.

Revision tags solve this problem. Revision tags are stable identifiers that point to revisions and can be used to avoid relabeling namespaces. Rather than relabeling the namespace, a mesh operator can simply change the tag to point to a new revision. All namespaces labeled with that tag will be updated at the same time.

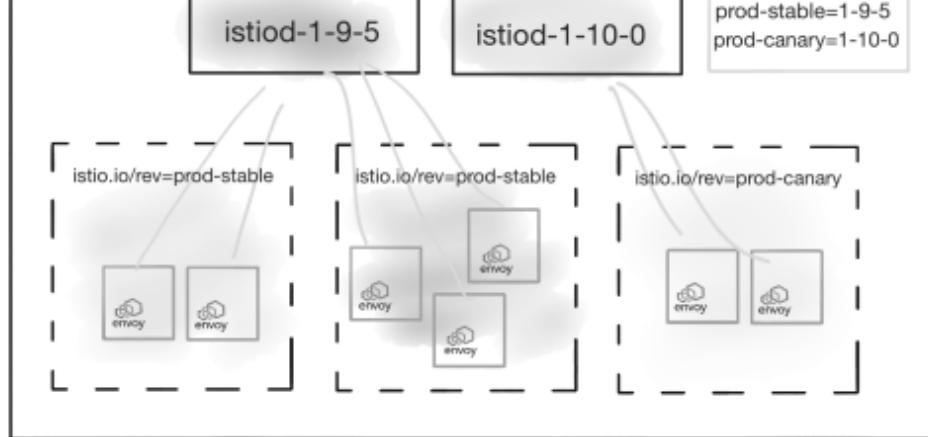
## Usage

Consider a cluster with two revisions installed, 1-9-5 and 1-10-0. The cluster operator creates a revision tag `prod-stable`, pointed at the older, stable 1-9-5 version, and a revision tag `prod-canary` pointed at the newer 1-10-0 revision. That state could be reached via these commands:

```
$ istioctl tag set prod-stable --revision 1-9-5  
$ istioctl tag set prod-canary --revision 1-10-0
```

The resulting mapping between revisions, tags, and namespaces is as shown below:





Two namespaces pointed to prod-stable and one pointed to prod-canary

The cluster operator can view this mapping in addition to tagged namespaces through the `istioctl`

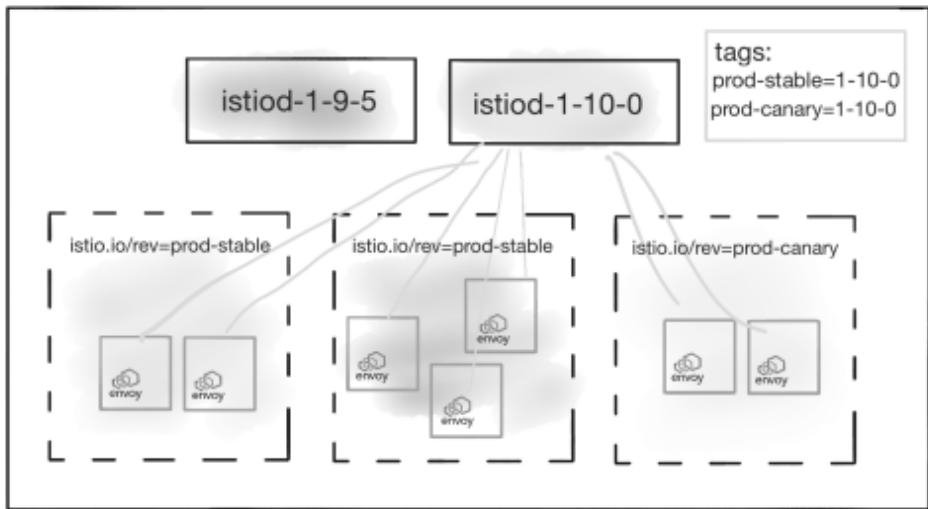
tag list command:

```
$ istioctl tag list  
TAG          REVISION NAMESPACES  
prod-canary 1-10-0    ...  
prod-stable 1-9-5    ...
```

After the cluster operator is satisfied with the stability of the control plane tagged with `prod-canary`, namespaces labeled `istio.io/rev=prod-stable` can be updated with one action by modifying the `prod-stable` revision tag to point to the newer `1-10-0` revision.

```
$ istioctl tag set prod-stable --revision 1-10-0
```

Now, the situation is as below:



Namespace labels unchanged but  
now all namespaces pointed to 1-  
10-0

Restarting injected workloads in the namespaces marked `prod-stable` will now result in those workloads using the `1-10-0` control plane. Notice that no namespace relabeling was required to migrate workloads to the new revision.

## Default tag

The revision pointed to by the tag `default` is considered the ***default revision*** and has additional semantic meaning.

The default revision will inject sidecars for the `istio-injection=enabled` namespace selector and `sidecar.istio.io/inject=true` object selector in addition to the `istio.io/rev=default` selectors. This makes it possible to migrate from using non-revisioned Istio to using a revision entirely without relabeling namespaces. To make a revision `1-10-0` the default, run:

```
$ istioctl tag set default --revision 1-10-0
```

When using the `default` tag alongside an existing non-revisioned Istio installation it is recommended to remove the old `MutatingWebhookConfiguration` (typically

called `istio-sidecar-injector`) to avoid having both the older and newer control planes attempt injection.

## Uninstall old control plane

After upgrading both the control plane and data plane, you can uninstall the old control plane. For example, the following command uninstalls a control plane of revision 1-6-5:

```
$ istioctl x uninstall --revision 1-6-5
```

If the old control plane does not have a revision label, uninstall it using its original installation options, for example:

```
$ istioctl x uninstall -f manifests/profiles/default.yaml
```

Confirm that the old control plane has been removed and only the new one still exists in the cluster:

```
$ kubectl get pods -n istio-system -l app=istiod
NAME                           READY   STATUS    RESTARTS   AG
E
istiod-canary-55887f699c-t8bh8   1/1     Running   0          27
m
```

Note that the above instructions only removed the resources for the specified control plane revision, but not cluster-scoped resources shared with other control planes. To uninstall Istio completely, refer to the [uninstall guide](#).

## **Uninstall canary control plane**

If you decide to rollback to the old control plane, instead of completing the canary upgrade, you can

uninstall the canary revision using `istioctl x uninstall --revision=canary`.

However, in this case you must first reinstall the gateway(s) for the previous revision manually, because the `uninstall` command will not automatically revert the previously in-place upgraded ones.

① Make sure to use the `istioctl` version corresponding to the old control plane to reinstall the old gateways and, to avoid downtime, make sure the old gateways are up and running before proceeding with the

canary uninstall.



# In-place Upgrades

⌚ 3 minute read page test

---

The `istioctl upgrade` command performs an upgrade of Istio. Before performing the upgrade, it checks that the Istio installation meets the upgrade eligibility criteria. Also, it alerts the user if it detects any changes in the profile default values between Istio versions.

**i** Canary Upgrade is safer than doing an in-place upgrade and is the recommended upgrade method.

The upgrade command can also perform a downgrade of Istio.

See the `istioctl upgrade` reference for all the options provided by the `istioctl upgrade` command.

`istioctl upgrade` is for in-place upgrade and

 not compatible with installations done with the `--revision` flag. Upgrades of such installations will fail with an error.

## Upgrade prerequisites

Before you begin the upgrade process, check the following prerequisites:

- The installed Istio version is no more than one

minor version less than the upgrade version. For example, 1.6.0 or higher is required before you start the upgrade process to 1.7.x.

- Your Istio installation was installed using `istioctl`.

## Upgrade steps

Traffic disruption may occur during the upgrade process. To minimize the disruption, ensure that at least two replicas of `istiod` are

running. Also, ensure that

PodDisruptionBudgets are configured with a minimum availability of 1.

The commands in this section should be run using the new version of `istioctl` which can be found in the `bin/` subdirectory of the downloaded package.

1. Download the new Istio release and change directory to the new release directory.
2. Ensure that your Kubernetes configuration points to the cluster to upgrade:

```
$ kubectl config view
```

3. Ensure that the upgrade is compatible with your environment.

```
$ istioctl x precheck
✓ No issues found when checking the cluster. Istio is safe
to install or upgrade!
To get started, check out https://istio.io/latest/docs/setup/getting-started/
```

4. Begin the upgrade by running this command:

```
$ istioctl upgrade
```

If you installed Istio using the `-f` flag, for example `istioctl install -f <IstioOperator-custom-resource-definition-file>`, then you must provide the same `-f` flag value to the `istioctl upgrade` command.



If you installed Istio using `--set` flags, ensure that you pass the same `--set` flags to upgrade, otherwise the customizations done with `--set` will be reverted. For production use, the use of a

configuration file instead of `--set` is recommended.

If you omit the `-f` flag, Istio upgrades using the default profile.

After performing several checks, `istioctl` will ask you to confirm whether to proceed.

5. `istioctl` will in-place upgrade the Istio control plane and gateways to the new version and indicate the completion status.
6. After `istioctl` completes the upgrade, you must manually update the Istio data plane by restarting any pods with Istio sidecars:

```
$ kubectl rollout restart deployment
```

# Downgrade prerequisites

Before you begin the downgrade process, check the following prerequisites:

- Your Istio installation was installed using `istioctl`.
- The Istio version you intend to downgrade to is no more than one minor version less than the installed Istio version. For example, you can

downgrade to no lower than 1.6.0 from Istio 1.7.x.

- Downgrade must be done using the `istioctl` binary version that corresponds to the Istio version that you intend to downgrade to. For example, if you are downgrading from Istio 1.7 to 1.6.5, use `istioctl` version 1.6.5.

## **Steps to downgrade to a lower Istio version**

You can use `istioctl upgrade` to downgrade to a lower version of Istio. The steps are identical to the upgrade process described in the previous section, only using the `istioctl` binary corresponding to the lower version (e.g., 1.6.5). When completed, Istio will be restored to the previously installed version.

Alternatively, `istioctl install` can be used to install an older version of the Istio control plane, but is not recommended because it does not perform any checks. For example, default values applied to the cluster for a configuration profile may change without warning.



# Upgrade with Helm

⌚ 6 minute read   ✖️ page test

---

Follow this guide to upgrade and configure an Istio mesh using Helm for in-depth evaluation. This guide assumes you have already performed an installation with Helm for a previous minor or patch version of Istio.

The Helm charts used in this guide are the same

underlying charts used when installing Istio via `Istioctl` or the Operator.

This feature is currently considered alpha.



Prior to Istio 1.9.0, installations using the Helm charts required hub and tag arguments: `--set global.hub="docker.io/istio"` and `--set global.tag="1.8.2"`. As of Istio 1.9.0 these are no longer required.

# Prerequisites

1. Download the Istio release.
2. Perform any necessary platform-specific setup.
3. Check the Requirements for Pods and Services.
4. Install a Helm client with a version higher than 3.1.1.



Helm 2 is not supported for installing Istio.

The commands in this guide use the Helm charts that

are included in the Istio release package located at  
manifests/charts.

## Upgrade steps

Change directory to the root of the release package  
and then follow the instructions below.

The default chart configuration uses the  
secure third party tokens for the service

account token projections used by Istio proxies to authenticate with the Istio control plane. Before proceeding to install any of the charts below, you should verify if third party tokens are enabled in your cluster by following the steps describe [here](#). If third party tokens are not enabled, you should add the option `--set global.jwtPolicy=first-party-jwt` to the Helm install commands. If the `jwtPolicy` is not set correctly, pods associated with `istiod`, gateways or workloads with injected Envoy proxies will not get deployed due to the missing `istio-token` volume.

Before upgrading Istio, it is recommended to run the `istioctl x precheck` command to make sure the upgrade is compatible with your environment.

```
$ istioctl x precheck
✓ No issues found when checking the cluster. Istio is safe to install or upgrade!
To get started, check out https://istio.io/latest/docs/setup/getting-started/
```

 Helm does not upgrade or delete CRDs **when performing an upgrade**. Because of this restriction, an additional step is required

when upgrading Istio with Helm.

## Create a backup

Before upgrading Istio in your cluster, we recommend creating a backup of your custom configurations, and restoring it from backup if necessary:

```
$ kubectl get istio-io --all-namespaces -oyaml > "$HOME"/istio_r  
esource_backup.yaml
```

You can restore your custom configuration like this:

```
$ kubectl apply -f "$HOME"/istio_resource_backup.yaml
```

## **Canary upgrade (recommended)**

You can install a canary version of Istio control plane to validate that the new version is compatible with your existing configuration and data plane using the steps below:

Note that when you install a canary version of the `istiod` service, the underlying cluster-wide resources from the base chart are shared across your primary and canary installations.

Currently, the support for canary upgrades for Istio ingress and egress gateways is actively in development and is considered experimental.

1. Upgrade the Kubernetes custom resource

## definitions (CRDs):

```
$ kubectl apply -f manifests/charts/base/crds
```

2. Install a canary version of the Istio discovery chart by setting the revision value:

```
$ helm install istiod-canary manifests/charts/istio-control  
/istio-discovery \  
--set revision=canary \  
-n istio-system
```

3. Verify that you have two versions of `istiod` installed in your cluster:

```
$ kubectl get pods -l app=istiod -L istio.io/rev -n istio-system
```

NAME	READY	STATUS	RESTART
S	AGE	REV	
istiod-5649c48ddc-dlk8h	1/1	Running	0
71m default			
istiod-canary-9cc9fd96f-jpc7n	1/1	Running	0
34m canary			

4. Follow the steps here to test or migrate existing workloads to use the canary control plane.
5. Once you have verified and migrated your workloads to use the canary control plane, you can uninstall your old control plane:

```
$ helm delete istiod -n istio-system
```

## 6. Upgrade the Istio base chart:

```
$ helm upgrade istio-base manifests/charts/base -n istio-system --skip-crds
```

# **Stable revision labels (experimental)**

 Stable revision labels are only supported when updating Istio from and to Istio versions 1.10+.

Manually relabeling namespaces when moving them to a new revision can be tedious and error-prone.

Revision tags solve this problem. Revision tags are stable identifiers that point to revisions and can be used to avoid relabeling namespaces. Rather than relabeling the namespace, a mesh operator can simply change the tag to point to a new revision. All namespaces labeled with that tag will be updated at the same time.

## Usage

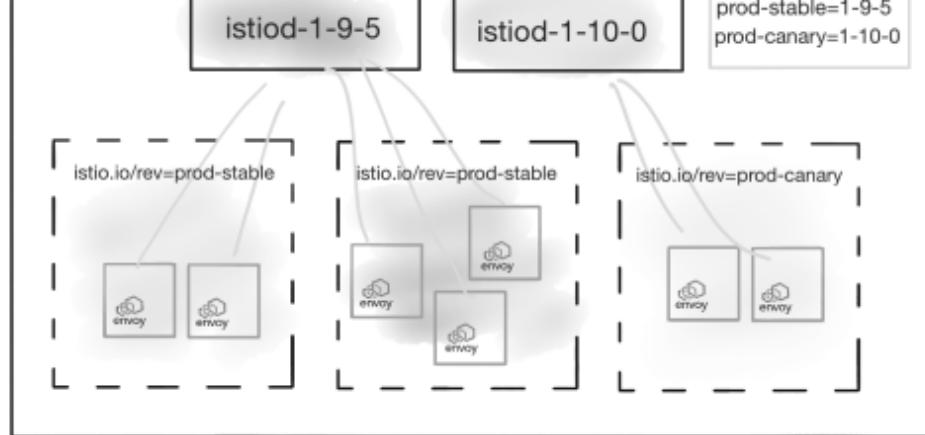
Consider a cluster with two revisions installed, 1-9-5 and 1-10-0. The cluster operator creates a revision tag prod-stable, pointed at the older, stable 1-9-5 version, and a revision tag prod-canary pointed at the newer 1-10-0 revision. That state could be reached via these commands:

```
$ helm template istiod manifests/charts/istio-control/istio-disc  
overv -s templates/revision-tags.yaml --set revisionTags={prod-s  
table} --set revision=1-9-5 -n istio-system | kubectl apply -f -  
$ helm template istiod manifests/charts/istio-control/istio-disc  
overv -s templates/revision-tags.yaml --set revisionTags={prod-c  
anary} --set revision=1-10-0 -n istio-system | kubectl apply -f -
```

These commands create new

MutatingWebhookConfiguration resources in your cluster, however, they are not owned by any Helm chart due to `kubectl` manually applying the templates. See the instructions below to uninstall revision tags.

The resulting mapping between revisions, tags, and namespaces is as shown below:



Two namespaces pointed to prod-stable and one pointed to prod-canary

The cluster operator can view this mapping in addition to tagged namespaces through the `istioctl`

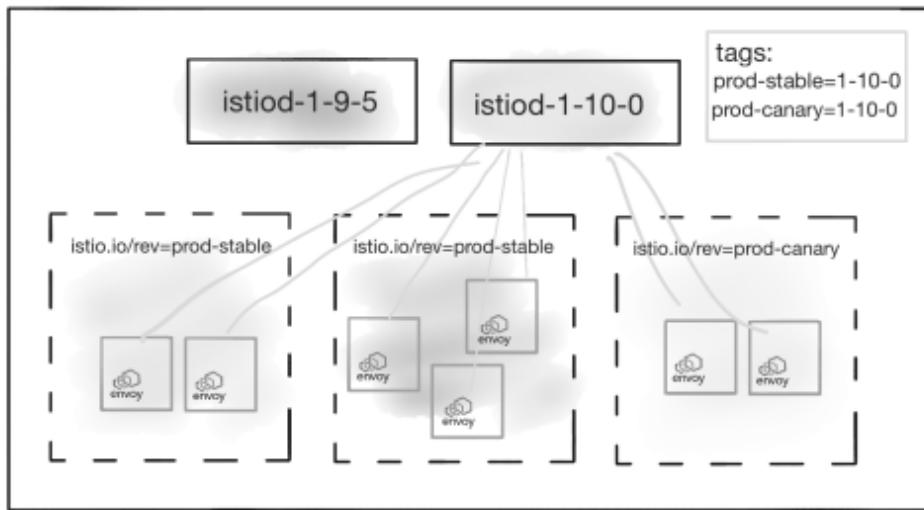
tag list command:

```
$ istioctl tag list  
TAG          REVISION NAMESPACES  
prod-canary 1-10-0    ...  
prod-stable 1-9-5    ...
```

After the cluster operator is satisfied with the stability of the control plane tagged with `prod-canary`, namespaces labeled `istio.io/rev=prod-stable` can be updated with one action by modifying the `prod-stable` revision tag to point to the newer `1-10-0` revision.

```
$ helm template istiod manifests/charts/istio-control/istio-discovery -s templates/revision-tags.yaml --set revisionTags={prod-stable} --set revision=1-10-0 -n istio-system | kubectl apply -f
```

Now, the situation is as below:



Namespace labels unchanged but  
now all namespaces pointed to 1-  
10-0

Restarting injected workloads in the namespaces marked `prod-stable` will now result in those workloads using the `1-10-0` control plane. Notice that no namespace relabeling was required to migrate workloads to the new revision.

## Default tag

The revision pointed to by the tag `default` is considered the ***default revision*** and has additional semantic meaning.

The default revision will inject sidecars for the `istio-injection=enabled` namespace selector and `sidecar.istio.io/inject=true` object selector in addition to the `istio.io/rev=default` selectors. This makes it possible to migrate from using non-revisioned Istio to using a revision entirely without relabeling namespaces. To make a revision `1-10-0` the default, run:

```
$ helm template istiod manifests/charts/istio-control/istio-discovery -s templates/revision-tags.yaml --set revisionTags={default} --set revision=1-10-0 -n istio-system | kubectl apply -f -
```

When using the `default` tag alongside an existing non-revisioned Istio installation it is recommended to remove the old `MutatingWebhookConfiguration` (typically called `istio-sidecar-injector`) to avoid having both the older and newer control planes attempt injection.

## In place upgrade

You can perform an in place upgrade of Istio in your

cluster using the Helm upgrade workflow.

This upgrade path is only supported from Istio version 1.8 and above.



Add your override values file or custom options to the commands below to preserve your custom configuration during Helm upgrades.

1. Upgrade the Kubernetes custom resource definitions (CRDs):

```
$ kubectl apply -f manifests/charts/base/crds
```

## 2. Upgrade the Istio base chart:

```
$ helm upgrade istio-base manifests/charts/base -n istio-system --skip-crds
```

## 3. Upgrade the Istio discovery chart:

```
$ helm upgrade istiod manifests/charts/istio-control/istio-discovery \
-n istio-system
```

## 4. (Optional) Upgrade the Istio ingress or egress gateway charts if installed in your cluster:

```
$ helm upgrade istio-ingress manifests/charts/gateways/istio-ingress \
    -n istio-system
$ helm upgrade istio-egress manifests/charts/gateways/istio-egress \
    -n istio-system
```

# Uninstall

Please refer to the [uninstall section](#) in our Helm install guide.

# Managing Gateways with Multiple Revisions

⌚ 5 minute read   ✎ page test

---

---



This feature is actively in development and is considered experimental.

With a single `IstioOperator` CR, any gateways defined in the CR (including the `istio-ingressgateway` installed in the default profile) are upgraded in place, even when the canary control plane method is used. This is undesirable because gateways are a critical component affecting application uptime. They should be upgraded last, after the new control and data plane versions are verified to be working.

This guide describes the recommended way to upgrade gateways by defining and managing them in a separate `IstioOperator` CR, separate from the one used to install and manage the control plane.

 To avoid problems with . (dot) not being a valid character in some Kubernetes paths, the revision name should not include . (dots).

## Istioctl

This section covers the installation and upgrade of a separate control plane and gateway using `istioctl`.

The example demonstrates how to upgrade Istio 1.8.0 to 1.8.1 using canary upgrade, with gateways being managed separately from the control plane.

## Installation with `istioctl`

1. Ensure that the main `IstioOperator` CR has a name and does not install a gateway:

```
# filename: control-plane.yaml
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
metadata:
  name: control-plane # REQUIRED
spec:
  profile: minimal
```

2. Create a separate `IstioOperator` CR for the gateway(s), ensuring that it has a name and has the `empty` profile:

```
# filename: gateways.yaml
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
metadata:
  name: gateways # REQUIRED
spec:
  profile: empty # REQUIRED
  components:
    ingressGateways:
      - name: istio-ingressgateway
        enabled: true
```

### 3. Install the CRS:

```
$ istio-1.8.0/bin/istioctl install -n istio-system -f control-plane.yaml --revision 1-8-0
$ istio-1.8.0/bin/istioctl install -n istio-system -f gateways.yaml --revision 1-8-0
```

Istioctl install and the operator track resource ownership through labels for both the revision and owning CR name. Only resources whose name and revision labels match the `IstioOperator` CR passed to `istioctl` install/operator will be affected by any changes to the CR - all other resources in the cluster will be ignored. It is important to make sure that each `IstioOperator` installs components that do not overlap with another `IstioOperator` CR, otherwise the two CR's will cause controllers or `istioctl` commands to interfere with each other.

# Upgrade with istioctl

Let's assume that the target version is 1.8.1.

1. Download the Istio 1.8.1 release and use the `istioctl` from that release to install the Istio 1.8.1 control plane:

```
$ istio-1.8.1/bin/istioctl install -f control-plane.yaml --  
revision 1-8-1
```

(Refer to the canary upgrade docs for more details on steps 2-4.)

2. Verify that the control plane is functional.

3. Label workload namespaces with `istio.io/rev=1-8-1` and restart the workloads.
4. Verify that the workloads are injected with the new proxy version and the cluster is functional.
5. At this point, the ingress gateway is still 1.8.0. You should see the following pods running:

```
$ kubectl get pods -n istio-system --show-labels
```

NAME	READY	STATUS	R
ESTARTS AGE LABELS			
istio-ingressgateway-65f8bdd46c-d49wf 21m	1/1	Running	0
		service.istio.io/canonical-revision=1-8-0 .	
..			
istiod-1-8-0-67f9b9b56-r22t5 22m	1/1	Running	0
		istio.io/rev=1-8-0 ...	
istiod-1-8-1-75dfd7d494-xhmbb 21s	1/1	Running	0
		istio.io/rev=1-8-1 ...	

As a last step, upgrade any gateways in the cluster to the new version:

```
$ istio-1.8.1/bin/istioctl install -f gateways.yaml --revision 1-8-1
```

## 6. Delete the 1.8.0 version of the control plane:

```
$ istio-1.8.1/bin/istioctl x uninstall --revision 1-8-0
```

# Operator

This section covers the installation and upgrade of a separate control plane and gateway using the Istio operator. The example demonstrates how to upgrade Istio 1.8.0 to 1.8.1 using canary upgrade, with gateways being managed separately from the control

plane.

# Installation with operator

1. Install the Istio operator with a revision into the cluster:

```
$ istio-1.8.0/bin/istioctl operator init --revision 1-8-0
```

2. Ensure that the main `IstioOperator` CR has a name and revision, and does not install a gateway:

```
# filename: control-plane-1-8-0.yaml
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
metadata:
  name: control-plane-1-8-0 # REQUIRED
spec:
  profile: minimal
  revision: 1-8-0 # REQUIRED
```

3. Create a separate `IstioOperator` CR for the gateway(s), ensuring that it has a name and has the `empty` profile:

```
# filename: gateways.yaml
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
metadata:
  name: gateways # REQUIRED
spec:
  profile: empty # REQUIRED
  revision: 1-8-0 # REQUIRED
  components:
    ingressGateways:
      - name: istio-ingressgateway
        enabled: true
```

4. Apply the files to the cluster with the following commands:

```
$ kubectl create namespace istio-system  
$ kubectl apply -n istio-system -f control-plane-1-8-0.yaml  
$ kubectl apply -n istio-system -f gateways.yaml
```

Verify that the operator and Istio control plane are installed and running.

## Upgrade with operator

Let's assume that the target version is 1.8.1.

1. Download the Istio 1.8.1 release and use the `istioctl` from that release to install the Istio 1.8.1

operator:

```
$ istio-1.8.1/bin/istioctl operator init --revision 1-8-1
```

2. Copy the control plane CR from the install step above as `control-plane-1-8-1.yaml`. Change all instances of `1-8-0` to `1-8-1` in the files.
3. Apply the new file to the cluster:

```
$ kubectl apply -n istio-system -f control-plane-1-8-1.yaml
```
4. Verify that two versions of `istiod` are running in the cluster. It may take several minutes for the operator to install the new control plane and for it to be in a running state.

```
$ kubectl -n istio-system get pod -l app=istiod
NAME                      READY   STATUS    RESTARTS
AGE
istiod-1-8-0-74f95c59c-4p6mc   1/1     Running   0
  68m
istiod-1-8-1-65b64fc749-5zq8w   1/1     Running   0
  13m
```

5. Refer to the canary upgrade docs for more details on rolling over workloads to the new Istio version:
  - Label workload namespaces with `istio.io/rev=1-8-1` and restart the workloads.
  - Verify that the workloads are injected with the new proxy version and the cluster is functional.

6. Upgrade the gateway to the new revision. Edit the `gateways.yaml` file from the installation step to change the revision from `1-8-0` to `1-8-1` and re-apply the file:

```
$ kubectl apply -n istio-system -f gateways.yaml
```

7. Verify that the gateway is running at version `1.8.1`.

```
$ kubectl -n istio-system get pod -l app=istio-ingressgateway --show-labels
NAME                               READY   STATUS    R
ESTARTS   AGE     LABELS
istio-ingressgateway-66dc957bd8-r2ptn   1/1     Running   0
                                         14m     app=istio-ingressgateway,service.istio.io/c
                                         canonical-revision=1-8-1...
```

## 8. Uninstall the old control plane:

```
$ kubectl delete istiooperator -n istio-system control-plan
e-1-8-0
```

## 9. Verify that only one version of `istiod` is running in the cluster.

```
$ kubectl -n istio-system get pod -l app=istiod
NAME                  READY   STATUS    RESTARTS   AGE
istiod-1-8-1-65b64fc749-5zq8w   1/1     Running   0          16m
```

# Installation Configuration Profiles

⌚ 2 minute read

---

---

This page describes the built-in configuration profiles that can be used when installing Istio. The profiles provide customization of the Istio control plane and of the sidecars for the Istio data plane.

You can start with one of Istio's built-in configuration profiles and then further customize the configuration for your specific needs. The following built-in configuration profiles are currently available:

1. **default**: enables components according to the default settings of the `IstioOperator` API. This profile is recommended for production deployments and for primary clusters in a multicloud mesh. You can display the default settings by running the `istioctl profile dump` command.
2. **demo**: configuration designed to showcase Istio

functionality with modest resource requirements. It is suitable to run the Bookinfo application and associated tasks. This is the configuration that is installed with the quick start instructions.



This profile enables high levels of tracing and access logging so it is not suitable for performance tests.

3. **minimal**: same as the default profile, but only the control plane components are installed. This allows you to configure the control plane and data

plane components (e.g., gateways) using separate profiles.

4. **external**: used for configuring a remote cluster that is managed by an external control plane or by a control plane in a primary cluster of a multicluster mesh.
5. **empty**: deploys nothing. This can be useful as a base profile for custom configuration.
6. **preview**: the preview profile contains features that are experimental. This is intended to explore new features coming to Istio. Stability, security, and performance are not guaranteed - use at your

own risk.

①

Some additional vendor-specific configuration profiles are also available. For more information, refer to the setup instructions for your platform.

The components marked as ✓ are installed within each profile:

default

demo

minimal

external

empty

pre

C  
o  
r  
e  
c  
o  
m  
p  
o  
n  
e  
n  
t  
s

✓

i  
s  
t  
i  
o  
-  
e  
g  
r  
e

s  
s  
g  
a  
t  
e  
w  
a  
y

✓

✓

✓

i

s  
t  
i  
o  
-  
i  
n  
g  
r  
e  
s  
s  
g

a  
t  
e  
w  
a  
y

✓

✓

✓

✓

i  
s  
t  
i



To further customize Istio, a number of addon components can also be installed. Refer to [integrations](#) for more details.



# Installing Gateways

⌚ 7 minute read ⚗ page test

---

Along with creating a service mesh, Istio allows you to manage gateways, which are Envoy proxies running at the edge of the mesh, providing fine-grained control over traffic entering and leaving the mesh.

Some of Istio's built in configuration profiles **deploy** gateways during installation. For example, a call to

`istioctl install` with default settings will deploy an ingress gateway along with the control plane. Although fine for evaluation and simple use cases, this couples the gateway to the control plane, making management and upgrade more complicated. For production Istio deployments, it is highly recommended to decouple these to allow independent operation.

Follow this guide to separately deploy and manage one or more gateways in a production installation of Istio.

# Prerequisites

This guide requires the Istio control plane to be installed before proceeding.

i

You can use the `minimal` profile, for example `istioctl install --set profile=minimal`, to prevent any gateways from being deployed during installation.

# Deploying a gateway

Using the same mechanisms as Istio sidecar injection, the Envoy proxy configuration for gateways can similarly be auto-injected.

Using auto-injection for gateway deployments is recommended as it gives developers full control over the gateway deployment, while also simplifying operations. When a new upgrade is available, or a configuration has changed, gateway pods can be updated by simply restarting them. This makes the experience of operating a gateway deployment the

same as operating sidecars.

To support users with existing deployment tools, Istio provides a few different ways to deploy a gateway. Each method will produce the same result. Choose the method you are most familiar with.

- i As a security best practice, it is recommended to deploy the gateway in a different namespace from the control plane.

IstioOperator

Helm

Kubernetes YAML

First, setup an IstioOperator configuration file, called ingress.yaml here:

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
metadata:
  name: ingress
spec:
  profile: empty # Do not install CRDs or the control plane
  components:
    ingressGateways:
      - name: ingressgateway
        namespace: istio-ingress
        enabled: true
        label:
          # Set a unique label for the gateway. This is required to ensure Gateways
```

```
# can select this workload
    istio: ingressgateway

values:
  gateways:
    istio-ingressgateway:
      # Enable gateway injection
      injectionTemplate: gateway
```

Then install using standard `istioctl` commands:

```
$ kubectl create namespace istio-ingress
$ istioctl install -f ingress.yaml
```

# Managing gateways

The following describes how to manage gateways after installation. For more information on their usage, follow the Ingress and Egress tasks.

## Gateway selectors

The labels on a gateway deployment's pods are used by `Gateway` configuration resources, so it's important that your `Gateway` selector matches these labels.

For example, in the above deployments, the `istio=ingressgateway` label is set on the gateway pods. To apply a `Gateway` to these deployments, you need to select the same label:

```
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  name: gateway
spec:
  selector:
    istio: ingressgateway
...
...
```

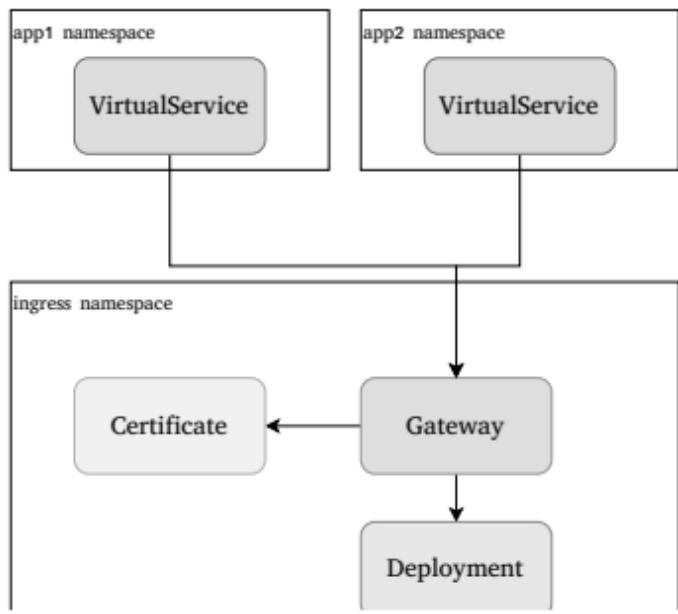
# Gateway deployment topologies

Depending on your mesh configuration and use cases, you may wish to deploy gateways in different ways. A few different gateway deployment patterns are shown below. Note that more than one of these patterns can be used within the same cluster.

## Shared gateway

In this model, a single centralized gateway is used by

many applications, possibly across many namespaces. Gateway(s) in the ingress namespace delegate ownership of routes to application namespaces, but retain control over TLS configuration.



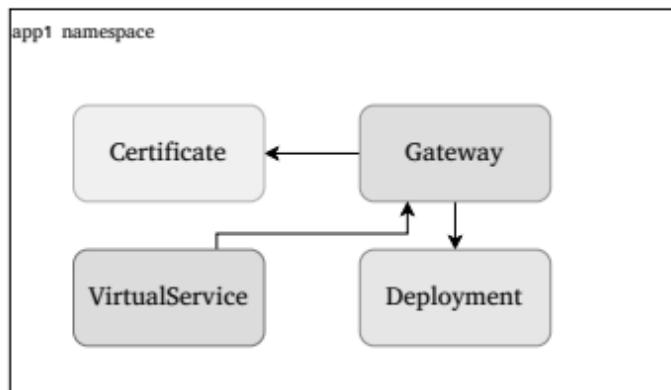
## Shared gateway

This model works well when you have many applications you want to expose externally, as they are able to use shared infrastructure. It also works well in use cases that have the same domain or TLS certificates shared by many applications.

## Dedicated application gateway

In this model, an application namespace has its own

dedicated gateway installation. This allows giving full control and ownership to a single namespace. This level of isolation can be helpful for critical applications that have strict performance or security requirements.



Dedicated application

## gateway

Unless there is another load balancer in front of Istio, this typically means that each application will have its own IP address, which may complicate DNS configurations.

## Upgrading gateways

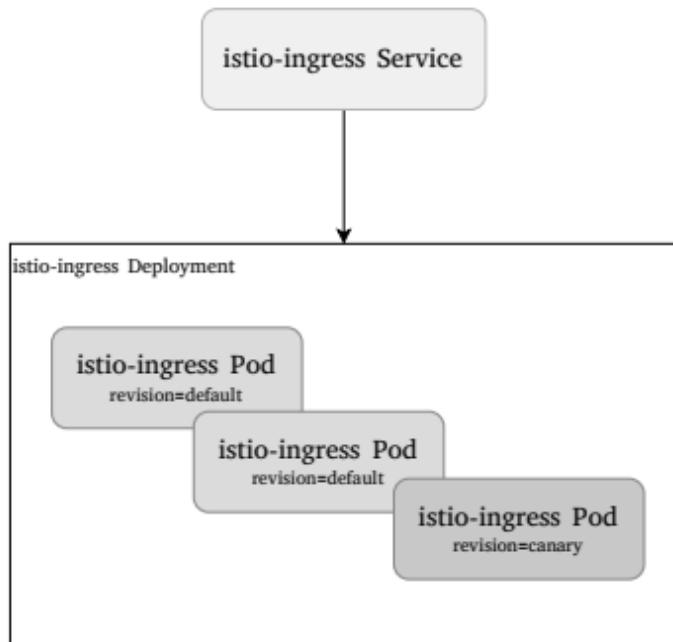
# In place upgrade

Because gateways utilize pod injection, new gateway pods that are created will automatically be injected with the latest configuration, which includes the version.

To pick up changes to the gateway configuration, the pods can simply be restarted, using commands such as `kubectl rollout restart deployment`.

If you would like to change the control plane revision in use by the gateway, you can set the `istio.io/rev` label

on the gateway Deployment, which will also trigger a rolling restart.



In place upgrade in

progress

## Canary upgrade (advanced)

This upgrade method depends on control plane revisions, and therefore can only be used in conjunction with control plane canary upgrade.



If you would like to more slowly control the rollout of

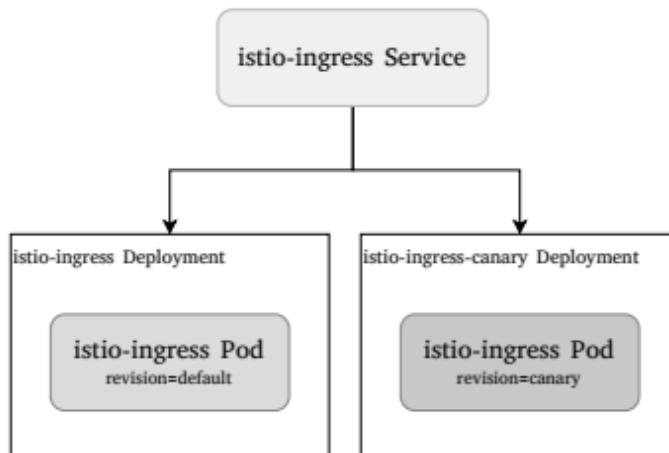
a new control plane revision, you can run multiple versions of a gateway deployment. For example, if you want to roll out a new revision, `canary`, create a copy of your gateway deployment with the `istio.io/rev=canary` label set:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: istio-ingressgateway-canary
  namespace: istio-ingress
spec:
  selector:
    matchLabels:
      istio: ingressgateway
template:
  metadata:
```

```
annotations:  
    inject.istio.io/templates: gateway  
labels:  
    istio: ingressgateway  
    istio.io/rev: canary # Set to the control plane revision  
you want to deploy  
spec:  
  containers:  
  - name: istio-proxy  
    image: auto
```

When this deployment is created, you will then have two versions of the gateway, both selected by the same Service:

```
$ kubectl get endpoints -o "custom-columns=NAME:.metadata.name,PODS:.subsets[*].addresses[*].targetRef.name"
NAME                PODS
istio-ingressgateway    istio-ingressgateway-788854c955-8gv96,istio-ingressgateway-canary-b78944cbd-mq2qf
```



Canary upgrade in  
progress

Unlike application services deployed inside the mesh, you cannot use Istio traffic shifting to distribute the traffic between the gateway versions because their traffic is coming directly from external clients that Istio does not control. Instead, you can control the distribution of traffic by the number of replicas of each deployment. If you use another load balancer in front of Istio, you may also use that to control the traffic distribution.

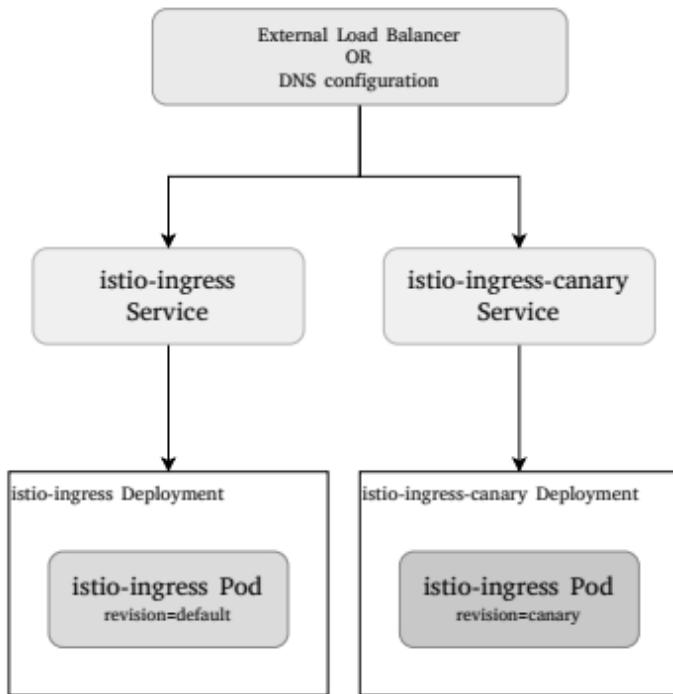
Because other installation methods bundle the gateway service, which controls its

external IP address, with the gateway Deployment, only the Kubernetes YAML method is supported for this upgrade method.

## **Canary upgrade with external traffic shifting (advanced)**

A variant of the canary upgrade approach is to shift the traffic between the versions using a high level

construct outside Istio, such as an external load balancer or DNS.



## Canary upgrade in progress with external traffic shifting

This offers fine-grained control, but may be unsuitable or overly complicated to set up in some environments.



# Installing the Sidecar

⌚ 6 minute read   ✖️ page test

---

## Injection

In order to take advantage of all of Istio's features, pods in the mesh must be running an Istio sidecar proxy.

The following sections describe two ways of injecting the Istio sidecar into a pod: enabling automatic Istio sidecar injection in the pod's namespace, or by manually using the `istioctl` command.

When enabled in a pod's namespace, automatic injection injects the proxy configuration at pod creation time using an admission controller.

Manual injection directly modifies configuration, like deployments, by adding the proxy configuration into it.

If you are not sure which one to use, automatic

injection is recommended.

## Automatic sidcar injection

Sidcars can be automatically added to applicable Kubernetes pods using a mutating webhook admission controller provided by Istio.



While admission controllers are enabled by default, some Kubernetes distributions may disable them. If this is the case, follow the

instructions to turn on admission controllers.

When you set the `istio-injection=enabled` label on a namespace and the injection webhook is enabled, any new pods that are created in that namespace will automatically have a sidecar added to them.

Note that unlike manual injection, automatic injection occurs at the pod-level. You won't see any change to the deployment itself. Instead, you'll want to check individual pods (via `kubectl describe`) to see the injected proxy.

# Deploying an app

Deploy sleep app. Verify both deployment and pod have a single container.

```
$ kubectl apply -f @samples/sleep/sleep.yaml@  
$ kubectl get deployment -o wide  
NAME      READY    UP-TO-DATE   AVAILABLE   AGE      CONTAINERS   IMAGE  
ES          SELECTOR  
sleep     1/1       1           1           12s     sleep        curl  
images/curl          app=sleep
```

```
$ kubectl get pod  
NAME          READY   STATUS    RESTARTS   AGE  
sleep-8f795f47d-hdcgs  1/1     Running   0          42s
```

Label the default namespace with istio-injection=enabled

```
$ kubectl label namespace default istio-injection=enabled --overwrite
$ kubectl get namespace -L istio-injection
NAME          STATUS  AGE    ISTIO-INJECTION
default        Active  5m9s  enabled
...
...
```

Injection occurs at pod creation time. Kill the running pod and verify a new pod is created with the injected sidecar. The original pod has 1/1 READY containers, and the pod with injected sidecar has 2/2 READY containers.

```
$ kubectl delete pod -l app=sleep  
$ kubectl get pod -l app=sleep  
pod "sleep-776b7bcdcd-7hpnk" deleted
```

NAME	READY	STATUS	RESTARTS	AGE
sleep-776b7bcdcd-7hpnk	1/1	Terminating	0	1m
sleep-776b7bcdcd-bhn9m	2/2	Running	0	7s

View detailed state of the injected pod. You should see the injected `istio-proxy` container and corresponding volumes.

```
$ kubectl describe pod -l app=sleep  
...  
Events:  
Type Reason Age From Message  
---- ---- -- -- -  
...  
...
```

	Normal	Created	11s	kubelet	Created container
istio-init	Normal	Started	11s	kubelet	Started container
istio-init					
	...				
	Normal	Created	10s	kubelet	Created container
sleep	Normal	Started	10s	kubelet	Started container
sleep					
	...				
	Normal	Created	9s	kubelet	Created container
istio-proxy	Normal	Started	8s	kubelet	Started container
istio-proxy					

Disable injection for the default namespace and verify new pods are created without the sidecar.

```
$ kubectl label namespace default istio-injection-  
$ kubectl delete pod -l app=sleep  
$ kubectl get pod  
namespace/default labeled  
pod "sleep-776b7bcdcd-bhn9m" deleted
```

NAME	READY	STATUS	RESTARTS	AGE
sleep-776b7bcdcd-bhn9m	2/2	Terminating	0	2m
sleep-776b7bcdcd-gmvnr	1/1	Running	0	2s

## Controlling the injection policy

In the above examples, you enabled and disabled injection at the namespace level. Injection can also be controlled on a per-pod basis, by configuring the `sidecar.istio.io/inject` label on a pod:

Resource	Label	Enabled value	Disabled value
Namespace	istio-injection	enabled	disabled
Pod	sidecar.istio.io/inject	"true"	"false"

If you are using control plane revisions, revision specific labels are instead used by a matching `istio.io/rev`

label. For example, for a revision named canary:

Resource	Enabled label	Disabled label
Namespace	istio.io/rev=canary	istio-injection=disabled
Pod	istio.io/rev=canary	sidecar.istio.io/inject="false"

If the `istio-injection` label and the `istio.io/rev` label are both present on the same namespace, the `istio-injection` label will take precedence.

The injector is configured with the following logic:

1. If either label is disabled, the pod is not injected
2. If either label is enabled, the pod is injected
3. If neither label is set, the pod is injected if  
`.values.sidecarInjectorWebhook.enableNamespacesByDefault` is enabled. This is not enabled by default, so generally this means the pod is not injected.

## Manual sidecar injection

To manually inject a deployment, use `istioctl kube-`

inject:

```
$ istioctl kube-inject -f @samples/sleep/sleep.yaml@ | kubectl apply -f -
serviceaccount/sleep created
service/sleep created
deployment.apps/sleep created
```

By default, this will use the in-cluster configuration.  
Alternatively, injection can be done using local copies  
of the configuration.

```
$ kubectl -n istio-system get configmap istio-sidecar-injector -o=jsonpath='{.data.config}' > inject-config.yaml  
$ kubectl -n istio-system get configmap istio-sidecar-injector -o=jsonpath='{.data.values}' > inject-values.yaml  
$ kubectl -n istio-system get configmap istio -o=jsonpath='{.data.mesh}' > mesh-config.yaml
```

Run kube-inject over the input file and deploy.

```
$ istioctl kube-inject \  
  --injectConfigFile inject-config.yaml \  
  --meshConfigFile mesh-config.yaml \  
  --valuesFile inject-values.yaml \  
  --filename @samples/sleep/sleep.yaml@ \  
  | kubectl apply -f -  
serviceaccount/sleep created  
service/sleep created  
deployment.apps/sleep created
```

Verify that the sidecar has been injected into the sleep pod with 2/2 under the READY column.

```
$ kubectl get pod -l app=sleep
NAME                  READY   STATUS    RESTARTS   AGE
sleep-64c6f57bc8-f5n4x 2/2     Running   0          24s
```

## Customizing injection

Generally, pods are injected based on the sidecar injection template, configured in the `istio-sidecar-injector` configmap. Per-pod configuration is available

to override these options on individual pods. This is done by adding an `istio-proxy` container to your pod. The sidecar injection will treat any configuration defined here as an override to the default injection template.

Care should be taken when customizing these settings, as this allows complete customization of the resulting Pod, including making changes that cause the sidecar container to not function properly.

For example, the following configuration customizes a variety of settings, including lowering the CPU requests, adding a volume mount, and adding a

## preStop hook:

```
apiVersion: v1
kind: Pod
metadata:
  name: example
spec:
  containers:
    - name: hello
      image: alpine
    - name: istio-proxy
      image: auto
      resources:
        requests:
          cpu: "100m"
  volumeMounts:
    - mountPath: /etc/certs
      name: certs
  lifecycle:
```

```
preStop:  
  exec:  
    command: ["sleep", "10"]  
volumes:  
- name: certs  
  secret:  
    secretName: istio-certs
```

In general, any field in a pod can be set. However, care must be taken for certain fields:

- Kubernetes requires the `image` field to be set before the injection has run. While you can set a specific image to override the default one, it is recommended to set the `image` to `auto` which will cause the sidecar injector to automatically select

the image to use.

- Some fields in Pod are dependent on related settings. For example, CPU request must be less than CPU limit. If both fields are not configured together, the pod may fail to start.

Additionally, certain fields are configurable by annotations on the pod, although it is recommended to use the above approach to customizing settings.

## **Custom templates (experimental)**

 This feature is experimental and subject to change, or removal, at any time.

Completely custom templates can also be defined at installation time. For example, to define a custom template that injects the `GREETING` environment variable into the `istio-proxy` container:

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
metadata:
  name: istio
spec:
  values:
    sidecarInjectorWebhook:
      templates:
        custom: |
          spec:
            containers:
              - name: istio-proxy
                env:
                  - name: GREETING
                    value: hello-world
```

Pods will, by default, use the `sidecar` injection template, which is automatically created. This can be

overridden by the `inject.istio.io/templates` annotation.  
For example, to apply the default template and our  
customization, you can set  
`inject.istio.io/templates=sidecar,custom.`

In addition to the `sidecar`, a `gateway` template is  
provided by default to support proxy injection into  
Gateway deployments.

# Customizing the installation configuration

⌚ 7 minute read

---

---

## Prerequisites

Before you begin, check the following prerequisites:

1. Download the Istio release.
2. Perform any necessary platform-specific setup.
3. Check the Requirements for Pods and Services.

In addition to installing any of Istio's built-in configuration profiles, `istioctl install` provides a complete API for customizing the configuration.

- The `IstioOperator` API

The configuration parameters in this API can be set individually using `--set` options on the command line.

For example, to enable debug logging in a default configuration profile, use this command:

```
$ istioctl install --set values.global.logging.level=debug
```

Alternatively, the `IstioOperator` configuration can be specified in a YAML file and passed to `istioctl` using the `-f` option:

```
$ istioctl install -f samples/operator/pilot-k8s.yaml
```

For backwards compatibility, the previous Helm installation options, with the exception of

Kubernetes resource settings, are also fully supported. To set them on the command line, prepend the option name with “values.”. For example, the following command overrides the `pilot.traceSampling` Helm configuration option:

① `$ istioctl install --set values.pilot.traceSampling=0.1`

Helm values can also be set in an IstioOperator CR (YAML file) as described in Customize Istio settings using the Helm API, **below**.

If you want to set Kubernetes resource settings, use the `IstioOperator` API as described in [Customize Kubernetes settings](#).

## Identify an Istio component

The `IstioOperator` API defines components as shown in the table below:

Components

base

pilot

ingressGateways

egressGateways

cni

istiodRemote

The configurable settings for each of these components are available in the API under [components](#).

<component name>. For example, to use the API to change (to false) the enabled setting for the pilot component, use --set components.pilot.enabled=false or set it in an IstioOperator resource like this:

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  components:
    pilot:
      enabled: false
```

All of the components also share a common API for changing Kubernetes-specific settings, under components.<component name>.k8s, as described in the

following section.

# Customize Kubernetes settings

The `IstioOperator` API allows each component's Kubernetes settings to be customized in a consistent way.

Each component has a `KubernetesResourceSpec`, which allows the following settings to be changed. Use this list to identify the setting to customize:

1. Resources
2. Readiness probes
3. Replica count
4. HorizontalPodAutoscaler
5. PodDisruptionBudget
6. Pod annotations
7. Service annotations
8. ImagePullPolicy
9. Priority class name
10. Node selector
11. Affinity and anti-affinity

- .2. Service
- .3. Toleration
- .4. Strategy
- .5. Env
- .6. Pod security context

All of these Kubernetes settings use the Kubernetes API definitions, so Kubernetes documentation can be used for reference.

The following example overlay file adjusts the resources and horizontal pod autoscaling settings for Pilot:

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  components:
    pilot:
      k8s:
        resources:
          requests:
            cpu: 1000m # override from default 500m
            memory: 4096Mi # ... default 2048Mi
        hpaSpec:
          maxReplicas: 10 # ... default 5
          minReplicas: 2 # ... default 1
```

Use `istioctl install` to apply the modified settings to the cluster:

```
$ istioctl install -f samples/operator/pilot-k8s.yaml
```

# Customize Istio settings using the Helm API

The `IstioOperator` API includes a pass-through interface to the Helm API using the `values` field.

The following YAML file configures global and Pilot settings through the Helm API:

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  values:
    pilot:
      traceSampling: 0.1 # override from 1.0
    global:
      monitoringPort: 15014
```

Some parameters will temporarily exist in both the Helm and `IstioOperator` APIs, including Kubernetes resources, namespaces and enablement settings. The Istio community recommends using the `IstioOperator` API as it is more consistent, is validated, and follows the community graduation process.

# Configure gateways

Gateways are a special type of component, since multiple ingress and egress gateways can be defined. In the `IstioOperator` API, gateways are defined as a list type. The `default` profile installs one ingress gateway, called `istio-ingressgateway`. You can inspect the default values for this gateway:

```
$ istioctl profile dump --config-path components.ingressGateways  
$ istioctl profile dump --config-path values.gateways.istio-ingr  
essgateway
```

These commands show both the `IstioOperator` and

Helm settings for the gateway, which are used together to define the generated gateway resources. The built-in gateways can be customized just like any other component.



From 1.7 onward, the gateway name must always be specified when overlaying. Not specifying any name no longer defaults to `istio-ingressgateway` **or** `istio-egressgateway`.

A new user gateway can be created by adding a new

list entry:

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  components:
    ingressGateways:
      - name: istio-ingressgateway
        enabled: true
      - namespace: user-ingressgateway-ns
        name: ilb-gateway
        enabled: true
    k8s:
      resources:
        requests:
          cpu: 200m
    serviceAnnotations:
      cloud.google.com/load-balancer-type: "internal"
    service:
```

```
ports:
  - port: 8060
    targetPort: 8060
    name: tcp-citadel-grpc-tls
  - port: 5353
    name: tcp-dns
```

Note that Helm values (`spec.values.gateways.istio-ingressgateway/egressgateway`) are shared by all ingress/egress gateways. If these must be customized per gateway, it is recommended to use a separate IstioOperator CR to generate a manifest for the user gateways, separate from the main Istio installation:

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  profile: empty
  components:
    ingressGateways:
      - name: ilb-gateway
        namespace: user-ingressgateway-ns
        enabled: true
        # Copy settings from istio-ingressgateway as needed.
  values:
    gateways:
      istio-ingressgateway:
        debug: error
```

# Advanced install

# customization

## Customizing external charts and profiles

The `istioctl install`, `manifest generate` and `profile` commands can use any of the following sources for charts and profiles:

- compiled in charts. This is the default if no `--manifests` option is set. The compiled in charts are

the same as those in the `manifests/` directory of the Istio release `.tgz`.

- charts in the local file system, e.g.,  
`istioctl install --manifests istio-1.11.3/manifests`
- charts in GitHub, e.g.,  
`istioctl install --manifests https://github.com/istio/istio/releases/download/1.1.3/istio-1.11.3-linux-arm64.tar.gz`

Local file system charts and profiles can be customized by editing the files in `manifests/`. For extensive changes, we recommend making a copy of the `manifests` directory and make changes there. Note, however, that the content layout in the `manifests`

directory must be preserved.

Profiles, found under `manifests/profiles/`, can be edited and new ones added by creating new files with the desired profile name and a `.yaml` extension. `istioctl` scans the `profiles` subdirectory and all profiles found there can be referenced by name in the `IstioOperatorSpec` `profile` field. Built-in profiles are overlaid on the default profile YAML before user overlays are applied. For example, you can create a new profile file called `custom1.yaml` which customizes some settings from the `default` profile, and then apply a user overlay file on top of that:

```
$ istioctl manifest generate --manifests mycharts/ --set profile=custom1 -f path-to-user-overlay.yaml
```

In this case, the `custom1.yaml` and `user-overlay.yaml` files will be overlaid on the `default.yaml` file to obtain the final values used as the input for manifest generation.

In general, creating new profiles is not necessary since a similar result can be achieved by passing multiple overlay files. For example, the command above is equivalent to passing two user overlay files:

```
$ istioctl manifest generate --manifests mycharts/ -f manifests/profiles/custom1.yaml -f path-to-user-overlay.yaml
```

Creating a custom profile is only required if you need to refer to the profile by name through the `IstioOperatorSpec`.

## Patching the output manifest

The `IstioOperator` CR, input to `istioctl`, is used to generate the output manifest containing the Kubernetes resources to be applied to the cluster. The output manifest can be further customized to add, modify or delete resources through the

IstioOperator overlays API, after it is generated but before it is applied to the cluster.

The following example overlay file (`patch.yaml`) demonstrates the type of output manifest patching that can be done:

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  profile: empty
  hub: docker.io/istio
  tag: 1.1.6
  components:
    pilot:
      enabled: true
      namespace: istio-control
```

```
k8s:  
  overlays:  
    - kind: Deployment  
      name: istiod  
      patches:  
        # Select list item by value  
        - path: spec.template.spec.containers.[name:discovery].args.[30m]  
          value: "60m" # overridden from 30m  
        # Select list item by key:value  
        - path: spec.template.spec.containers.[name:discovery].ports.[containerPort:8080].containerPort  
          value: 1234  
        # Override with object (note | on value: first line)  
        - path: spec.template.spec.containers.[name:discovery].env.[name:POD_NAMESPACE].valueFrom  
          value: |  
          fieldRef:  
            apiVersion: v2
```

```
        fieldPath: metadata.myPath
    # Deletion of list item
    - path: spec.template.spec.containers.[name:discovery].env.[name:REVISION]
        # Deletion of map item
        - path: spec.template.spec.containers.[name:discovery].securityContext
            - kind: Service
                name: istiod
                patches:
                    - path: spec.ports.[name:https-dns].port
                        value: 11111 # OVERRIDDEN
```

Passing the file to `istioctl manifest generate -f patch.yaml` applies the above patches to the default profile output manifest. The two patched resources will be modified as shown below (some parts of the

resources are omitted for brevity):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: istiod
spec:
  template:
    spec:
      containers:
        - args:
            - 60m
          env:
            - name: POD_NAMESPACE
              valueFrom:
                fieldRef:
                  apiVersion: v2
                  fieldPath: metadata.myPath
        name: discovery
```

```
  ports:  
    - containerPort: 1234  
---  
apiVersion: v1  
kind: Service  
metadata:  
  name: istiod  
spec:  
  ports:  
    - name: https-dns  
      port: 11111  
---
```

Note that the patches are applied in the given order. Each patch is applied over the output from the previous patch. Paths in patches that don't exist in the output manifest will be created.

# List item path selection

Both the `istioctl --set` flag and the `k8s.overlays` field in `IstioOperator` CR support list item selection by `[index]`, `[value]` or by `[key:value]`. The `-set` flag also creates any intermediate nodes in the path that are missing in the resource.

# Install Istio with the Istio CNI plugin

⌚ 7 minute read ✓ page test

---

Follow this guide to install, configure, and use an Istio mesh using the Istio Container Network Interface (CNI) plugin.

By default Istio injects an init container, `istio-init`, in pods deployed in the mesh. The `istio-init` container

sets up the pod network traffic redirection to/from the Istio sidecar proxy. This requires the user or service-account deploying pods to the mesh to have sufficient Kubernetes RBAC permissions to deploy containers with the `NET_ADMIN` and `NET_RAW` capabilities.

Requiring Istio users to have elevated Kubernetes RBAC permissions is problematic for some organizations' security compliance. The Istio CNI plugin is a replacement for the `istio-init` container that performs the same networking functionality but without requiring Istio users to enable elevated Kubernetes RBAC permissions.

The Istio CNI plugin identifies user application pods

with sidecars requiring traffic redirection and sets this up in the Kubernetes pod lifecycle's network setup phase, thereby removing the requirement for the NET\_ADMIN and NET\_RAW capabilities for users deploying pods into the Istio mesh. The Istio CNI plugin replaces the functionality provided by the `istio-init` container.



Note: The Istio CNI plugin operates as a chained CNI plugin, and it is designed to be used with another CNI plugin, such as PTP or Calico. See compatibility with other CNI plugins for

details.

# Install CNI

## Prerequisites

1. Install Kubernetes with the container runtime supporting CNI and `kubelet` configured with the main CNI plugin enabled via `--network-plugin=cni`.

- AWS EKS, Azure AKS, and IBM Cloud IKS clusters have this capability.
  - Google Cloud GKE clusters have CNI enabled when any of the following features are **enabled**: network policy, intranode visibility, workload identity, pod security policy, **or** dataplane v2.
  - OpenShift has CNI enabled by default.
2. Install Kubernetes with the ServiceAccount admission controller **enabled**.
- The Kubernetes documentation highly recommends this for all Kubernetes

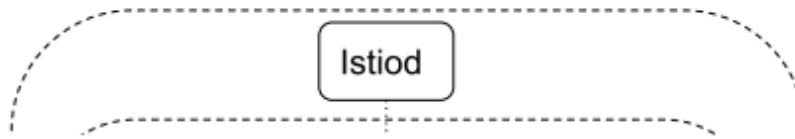
installations where ServiceAccounts are utilized.

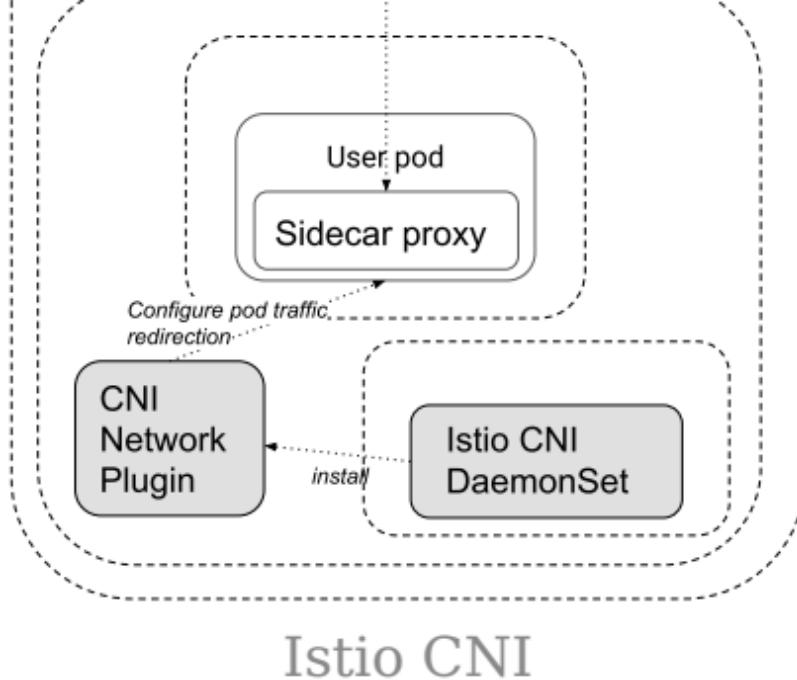
## Install Istio with CNI plugin

In most environments, a basic Istio cluster with CNI enabled can be installed using the following configuration:

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  components:
    cni:
      enabled: true
```

This will deploy an `istio-cni-node` DaemonSet into the cluster, which installs the Istio CNI plugin binary to each node and sets up the necessary configuration for the plugin. The CNI DaemonSet runs with `system-node-critical` PriorityClass.





There are several commonly used install options:

- `components.cni.namespace=kube-system` configures the namespace to install the CNI DaemonSet.
- `values.cni.cniBinDir` and `values.cni.cniConfDir` configure the directory paths to install the plugin binary and create plugin configuration.  
`values.cni.cniConfFileName` configures the name of the plugin configuration file.
- `values.cni.chained` controls whether to configure the plugin as a chained CNI plugin.

There is a time gap between a node becomes schedulable and the Istio CNI plugin

becomes ready on that node. If an application pod starts up during this time, it is possible that traffic redirection is not properly set up and traffic would be able to bypass the Istio sidecar. This race condition is mitigated by a “detect and repair” method. Please take a look at race condition & mitigation section to understand the implication of this mitigation.

## Hosted Kubernetes settings

The `istio-cni` plugin is expected to work with any hosted Kubernetes version using CNI plugins. The default installation configuration works with most platforms. Some platforms required special installation settings.

- Google Kubernetes Engine

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  components:
    cni:
      enabled: true
      namespace: kube-system
  values:
    cni:
      cniBinDir: /home/kubernetes/bin
```

- Red Hat OpenShift 4.2+

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  components:
    cni:
      enabled: true
      namespace: kube-system
  values:
    sidecarInjectorWebhook:
      injectedAnnotations:
        k8s.v1.cni.cncf.io/networks: istio-cni
    cni:
      cniBinDir: /var/lib/cni/bin
      cniConfDir: /etc/cni/multus/net.d
      cniConfFileName: istio-cni.conf
      chained: false
```

# Operation details

## Upgrade

When upgrading Istio with in-place upgrade, the CNI component can be upgraded together with the control plane using one `IstioOperator` resource.

When upgrading Istio with canary upgrade, because the CNI component runs as a cluster singleton, it is recommended to operate and upgrade the CNI component separately from the revised control

plane. The following `IstioOperator` can be used to operate the CNI component independently.

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  profile: empty # Do not include other components
  components:
    cni:
      enabled: true
  values:
    cni:
      excludeNamespaces:
        - istio-system
        - kube-system
```

When installing revisioned control planes with the

CNI component enabled, `values.istio_cni.enabled` needs to be set, so that sidecar injector does not inject the `istio-init` init container.

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  revision: REVISION_NAME
  ...
  values:
    istio_cni:
      enabled: true
  ...
```

The CNI plugin at version `1.x` is compatible with control plane at version `1.x-1`, `1.x`, and `1.x+1`, which

means CNI and control plane can be upgraded in any order, as long as their version difference is within one minor version.

## Race condition & mitigation

The Istio CNI DaemonSet installs the CNI network plugin on every node. However, a time gap exists between when the DaemonSet pod gets scheduled onto a node, and the CNI plugin is installed and ready to be used. There is a chance that an application pod

starts up during that time gap, and the kubelet has no knowledge of the Istio CNI plugin. The result is that the application pod comes up without Istio traffic redirection and bypasses Istio sidecar.

To mitigate the race between an application pod and the Istio CNI DaemonSet, an `istio-validation` init container is added as part of the sidecar injection, which detects if traffic redirection is set up correctly, and blocks the pod starting up if not. The CNI DaemonSet will detect and evict any pod stuck in such state. When the new pod starts up, it should have traffic redirection set up properly. This mitigation is enabled by default and can be turned off

by setting `values.cni.repair.enabled` to false.

# Traffic redirection parameters

To redirect traffic in the application pod's network namespace to/from the Istio proxy sidecar, the Istio CNI plugin configures the namespace's iptables. You can adjust traffic redirection parameters using the same pod annotations as normal, such as ports and IP ranges to be included or excluded from redirection. See resource annotations for available parameters.

# Compatibility with application init containers

The Istio CNI plugin may cause networking connectivity problems for any application init containers. When using Istio CNI, kubelet starts an injected pod with the following steps:

1. The Istio CNI plugin sets up traffic redirection to the Istio sidecar proxy within the pod.
2. All init containers execute and complete

successfully.

3. The Istio sidecar proxy starts in the pod along with the pod's other containers.

Init containers execute before the sidecar proxy starts, which can result in traffic loss during their execution. Avoid this traffic loss with one or both of the following settings:

- Set the `traffic.sidecar.istio.io/excludeOutboundIPRanges` annotation to disable redirecting traffic to any CIDRs the init containers communicate with.

- Set the `traffic.sidecar.istio.io/excludeOutboundPorts` annotation to disable redirecting traffic to the specific outbound ports the init containers use.

 Please use the above settings with caution, since the IP/port exclusion annotations not only apply to init container traffic, but also application container traffic. i.e. application traffic sent to the configured IP/port will bypass the Istio sidecar.

# Compatibility with other CNI plugins

The Istio CNI plugin maintains compatibility with the same set of CNI plugins as the current `istio-init` container which requires the `NET_ADMIN` and `NET_RAW` capabilities.

The Istio CNI plugin operates as a chained CNI plugin. This means its configuration is added to the existing CNI plugins configuration as a new configuration list element. See the CNI specification reference for further details. When a pod is created or

deleted, the container runtime invokes each plugin in the list in order. The Istio CNI plugin only performs actions to setup the application pod's traffic redirection to the injected Istio proxy sidecar (using iptables in the pod's network namespace).



The Istio CNI plugin should not interfere with the operations of the base CNI plugin that configures the pod's networking setup, although not all CNI plugins have been validated.



# Request Routing

⌚ 5 minute read ✓ page test

---

This task shows you how to route requests dynamically to multiple versions of a microservice.

## Before you begin

- Setup Istio by following the instructions in the Installation guide.
- Deploy the Bookinfo sample application.
- Review the Traffic Management concepts doc. Before attempting this task, you should be familiar with important terms such as *destination rule*, *virtual service*, and *subset*.

## About this task

The Istio Bookinfo sample consists of four separate microservices, each with multiple versions. Three different versions of one of the microservices, reviews, have been deployed and are running concurrently. To illustrate the problem this causes, access the Bookinfo app's /productpage in a browser and refresh several times. You'll notice that sometimes the book review output contains star ratings and other times it does not. This is because without an explicit default service version to route to, Istio routes requests to all available versions in a round robin fashion.

The initial goal of this task is to apply rules that route all traffic to v1 (version 1) of the microservices. Later,

you will apply a rule to route traffic based on the value of an HTTP request header.

## Apply a virtual service

To route to one version only, you apply virtual services that set the default version for the microservices. In this case, the virtual services will route all traffic to `v1` of each microservice.

If you haven't already applied destination rules, follow the instructions in [Apply Default Destination Rules](#).

1. Run the following command to apply the virtual services:

```
$ kubectl apply -f @samples/bookinfo/networking/virtual-service-all-v1.yaml@
```

Because configuration propagation is eventually consistent, wait a few seconds for the virtual

services to take effect.

2. Display the defined routes with the following command:

```
$ kubectl get virtualservices -o yaml
- apiVersion: networking.istio.io/v1beta1
  kind: VirtualService
  ...
  spec:
    hosts:
      - details
    http:
      - route:
          - destination:
              host: details
              subset: v1
- apiVersion: networking.istio.io/v1beta1
  kind: VirtualService
```

```
...
spec:
  hosts:
    - productpage
  http:
    - route:
        - destination:
            host: productpage
            subset: v1
- apiVersion: networking.istio.io/v1beta1
kind: VirtualService
...
spec:
  hosts:
    - ratings
  http:
    - route:
        - destination:
            host: ratings
            subset: v1
```

```
- apiVersion: networking.istio.io/v1beta1
  kind: VirtualService
  ...
  spec:
    hosts:
      - reviews
    http:
      - route:
          - destination:
              host: reviews
              subset: v1
```

3. You can also display the corresponding `subset` definitions with the following command:

```
$ kubectl get destinationrules -o yaml
```

You have configured Istio to route to the `v1` version of

the Bookinfo microservices, most importantly the reviews service version 1.

## Test the new routing configuration

You can easily test the new configuration by once again refreshing the `/productpage` of the Bookinfo app.

1. Open the Bookinfo site in your browser. The URL is `http://$GATEWAY_URL/productpage`, where

`$GATEWAY_URL` is the External IP address of the ingress, as explained in the Bookinfo doc.

Notice that the reviews part of the page displays with no rating stars, no matter how many times you refresh. This is because you configured Istio to route all traffic for the reviews service to the version `reviews:v1` and this version of the service does not access the star ratings service.

You have successfully accomplished the first part of this task: route traffic to one version of a service.

# Route based on user identity

Next, you will change the route configuration so that all traffic from a specific user is routed to a specific service version. In this case, all traffic from a user named Jason will be routed to the service `reviews:v2`.

Note that Istio doesn't have any special, built-in understanding of user identity. This example is enabled by the fact that the `productpage` service adds a custom `end-user` header to all outbound HTTP requests to the `reviews` service.

Remember, reviews:v2 is the version that includes the star ratings feature.

1. Run the following command to enable user-based routing:

```
$ kubectl apply -f @samples/bookinfo/networking/virtual-service-reviews-test-v2.yaml@
```

2. Confirm the rule is created:

```
$ kubectl get virtualservice reviews -o yaml
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
...
spec:
```

```
hosts:
- reviews

http:
- match:
  - headers:
    end-user:
      exact: jason

route:
- destination:
  host: reviews
  subset: v2
- route:
  - destination:
    host: reviews
    subset: v1
```

3. On the /productpage of the Bookinfo app, log in as user jason.

Refresh the browser. What do you see? The star ratings appear next to each review.

4. Log in as another user (pick any name you wish).

Refresh the browser. Now the stars are gone.

This is because traffic is routed to `reviews:v1` for all users except Jason.

You have successfully configured Istio to route traffic based on user identity.

## Understanding what

# happened

In this task, you used Istio to send 100% of the traffic to the `v1` version of each of the Bookinfo services. You then set a rule to selectively send traffic to version `v2` of the `reviews` service based on a custom end-user header added to the request by the `productpage` service.

Note that Kubernetes services, like the Bookinfo ones used in this task, must adhere to certain restrictions to take advantage of Istio's L7 routing features. Refer to the Requirements for Pods and Services for details.

In the traffic shifting task, you will follow the same basic pattern you learned here to configure route rules to gradually send traffic from one version of a service to another.

## Cleanup

1. Remove the application virtual services:

```
$ kubectl delete -f @samples/bookinfo/networking/virtual-service-all-v1.yaml@
```

2. If you are not planning to explore any follow-on tasks, refer to the Bookinfo cleanup instructions to shutdown the application.

# Fault Injection

⌚ 5 minute read ✓ page test

---

This task shows you how to inject faults to test the resiliency of your application.

## Before you begin

- Set up Istio by following the instructions in the Installation guide.
- Deploy the Bookinfo sample application including the default destination rules.
- Review the fault injection discussion in the Traffic Management concepts doc.
- Apply application version routing by either performing the request routing task or by running the following commands:

```
$ kubectl apply -f @samples/bookinfo/networking/virtual-service-all-v1.yaml@  
$ kubectl apply -f @samples/bookinfo/networking/virtual-service-reviews-test-v2.yaml@
```

- With the above configuration, this is how requests flow:
  - productpage → reviews:v2 → ratings (only for user jason)
  - productpage → reviews:v1 (for everyone else)

## Injecting an HTTP delay fault

To test the Bookinfo application microservices for

resiliency, inject a 7s delay between the reviews:v2 and ratings microservices for user jason. This test will uncover a bug that was intentionally introduced into the Bookinfo app.

Note that the reviews:v2 service has a 10s hard-coded connection timeout for calls to the ratings service. Even with the 7s delay that you introduced, you still expect the end-to-end flow to continue without any errors.

1. Create a fault injection rule to delay traffic coming from the test user jason.

```
$ kubectl apply -f @samples/bookinfo/networking/virtual-service-ratings-test-delay.yaml@
```

## 2. Confirm the rule was created:

```
$ kubectl get virtualservice ratings -o yaml
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
...
spec:
  hosts:
  - ratings
  http:
  - fault:
      delay:
        fixedDelay: 7s
        percentage:
          value: 100
  match:
```

```
- headers:  
    end-user:  
        exact: jason  
route:  
- destination:  
    host: ratings  
    subset: v1  
- route:  
- destination:  
    host: ratings  
    subset: v1
```

Allow several seconds for the new rule to propagate to all pods.

# Testing the delay configuration

1. Open the Bookinfo web application in your browser.
2. On the /productpage web page, log in as user jason.

You expect the Bookinfo home page to load without errors in approximately 7 seconds.

However, there is a problem: the Reviews section displays an error message:

Sorry, product reviews are currently unavailable for this book.

3. View the web page response times:
  1. Open the *Developer Tools* menu in your web browser.
  2. Open the Network tab
  3. Reload the /productpage web page. You will see that the page actually loads in about 6 seconds.

# Understanding what happened

You've found a bug. There are hard-coded timeouts in the microservices that have caused the `reviews` service to fail.

As expected, the 7s delay you introduced doesn't affect the `reviews` service because the timeout between the `reviews` and `ratings` service is hard-coded at 10s. However, there is also a hard-coded timeout between the `productpage` and the `reviews` service, coded as  $3\text{s} + 1$  retry for 6s total. As a result, the `productpage`

call to reviews times out prematurely and throws an error after 6s.

Bugs like this can occur in typical enterprise applications where different teams develop different microservices independently. Istio's fault injection rules help you identify such anomalies without impacting end users.

 Notice that the fault injection test is restricted to when the logged in user is jason. If you login as any other user, you will not experience any delays.

# Fixing the bug

You would normally fix the problem by:

1. Either increasing the productpage to reviews service timeout or decreasing the reviews to ratings timeout
2. Stopping and restarting the fixed microservice
3. Confirming that the /productpage web page returns

its response without any errors.

However, you already have a fix running in v3 of the reviews service. The `reviews:v3` service reduces the reviews to ratings timeout from 10s to 2.5s so that it is compatible with (less than) the timeout of the downstream `productpage` requests.

If you migrate all traffic to `reviews:v3` as described in the traffic shifting task, you can then try to change the delay rule to any amount less than 2.5s, for example 2s, and confirm that the end-to-end flow continues without any errors.

# Injecting an HTTP abort fault

Another way to test microservice resiliency is to introduce an HTTP abort fault. In this task, you will introduce an HTTP abort to the `ratings` microservices for the test user `jason`.

In this case, you expect the page to load immediately and display the `Ratings service is currently unavailable` message.

1. Create a fault injection rule to send an HTTP

abort for user jason:

```
$ kubectl apply -f @samples/bookinfo/networking/virtual-service-ratings-test-abort.yaml@
```

## 2. Confirm the rule was created:

```
$ kubectl get virtualservice ratings -o yaml
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
...
spec:
  hosts:
    - ratings
  http:
    - fault:
        abort:
          httpStatus: 500
          percentage:
```

```
    value: 100
match:
- headers:
    end-user:
        exact: jason
route:
- destination:
    host: ratings
    subset: v1
- route:
    - destination:
        host: ratings
        subset: v1
```

# Testing the abort

# configuration

1. Open the Bookinfo web application in your browser.
2. On the /productpage, log in as user jason.

If the rule propagated successfully to all pods, the page loads immediately and the Ratings service is currently unavailable message appears.

3. If you log out from user jason or open the Bookinfo application in an anonymous window (or in another browser), you will see that /productpage still calls reviews:v1 (which does not call ratings at

all) for everybody but `jason`. Therefore you will not see any error message.

# Cleanup

1. Remove the application routing rules:

```
$ kubectl delete -f @samples/bookinfo/networking/virtual-service-all-v1.yaml@
```

2. If you are not planning to explore any follow-on tasks, refer to the Bookinfo cleanup instructions to

shutdown the application.



# Traffic Shifting

⌚ 3 minute read ✓ page test

---

This task shows you how to shift traffic from one version of a microservice to another.

A common use case is to migrate traffic gradually from an older version of a microservice to a new one. In Istio, you accomplish this goal by configuring a sequence of routing rules that redirect a percentage

of traffic from one destination to another.

In this task, you will use send 50% of traffic to reviews:v1 and 50% to reviews:v3. Then, you will complete the migration by sending 100% of traffic to reviews:v3.

## Before you begin

- Setup Istio by following the instructions in the Installation guide.

- Deploy the Bookinfo sample application.
- Review the Traffic Management concepts doc.

## Apply weight-based routing



If you haven't already applied destination rules, follow the instructions in [Apply Default Destination Rules](#).

1. To get started, run this command to route all traffic to the v1 version of each microservice.

```
$ kubectl apply -f @samples/bookinfo/networking/virtual-service-all-v1.yaml@
```

2. Open the Bookinfo site in your browser. The URL is `http://$GATEWAY_URL/productpage`, where `$GATEWAY_URL` is the External IP address of the ingress, as explained in the Bookinfo doc.

Notice that the reviews part of the page displays with no rating stars, no matter how many times you refresh. This is because you configured Istio to route all traffic for the reviews service to the

version reviews:v1 and this version of the service does not access the star ratings service.

3. Transfer 50% of the traffic from reviews:v1 to reviews:v3 with the following command:

```
$ kubectl apply -f @samples/bookinfo/networking/virtual-service-reviews-50-v3.yaml@
```

Wait a few seconds for the new rules to propagate.

4. Confirm the rule was replaced:

```
$ kubectl get virtualservice reviews -o yaml
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
...
spec:
  hosts:
    - reviews
  http:
    - route:
        - destination:
            host: reviews
            subset: v1
            weight: 50
        - destination:
            host: reviews
            subset: v3
            weight: 50
```

5. Refresh the /productpage in your browser and you

now see *red* colored star ratings approximately 50% of the time. This is because the `v3` version of reviews accesses the star ratings service, but the `v1` version does not.

With the current Envoy sidecar implementation, you may need to refresh the `/productpage` many times -perhaps 15 or more-to see the proper distribution.

 You can modify the rules to route 90% of the traffic to `v3` to see red stars more often.

6. Assuming you decide that the reviews:v3 microservice is stable, you can route 100% of the traffic to reviews:v3 by applying this virtual service:

```
$ kubectl apply -f @samples/bookinfo/networking/virtual-service-reviews-v3.yaml@
```

Now when you refresh the /productpage you will always see book reviews with *red* colored star ratings for each review.

# Understanding what happened

In this task you migrated traffic from an old to new version of the `reviews` service using Istio's weighted routing feature. Note that this is very different than doing version migration using the deployment features of container orchestration platforms, which use instance scaling to manage the traffic.

With Istio, you can allow the two versions of the `reviews` service to scale up and down independently, without affecting the traffic distribution between

them.

For more information about version routing with autoscaling, check out the blog article [Canary Deployments using Istio](#).

## Cleanup

1. Remove the application routing rules:

```
$ kubectl delete -f @samples/bookinfo/networking/virtual-service-all-v1.yaml@
```

2. If you are not planning to explore any follow-on tasks, refer to the Bookinfo cleanup instructions to shutdown the application.

# TCP Traffic Shifting

⌚ 4 minute read ✓ page test

---

This task shows you how to shift TCP traffic from one version of a microservice to another.

A common use case is to migrate TCP traffic gradually from an older version of a microservice to a new one. In Istio, you accomplish this goal by configuring a sequence of routing rules that redirect

a percentage of TCP traffic from one destination to another.

In this task, you will send 100% of the TCP traffic to `tcp-echo:v1`. Then, you will route 20% of the TCP traffic to `tcp-echo:v2` using Istio's weighted routing feature.

## Before you begin

- Setup Istio by following the instructions in the

Installation guide.

- Review the Traffic Management concepts doc.

## Set up the test environment

1. To get started, create a namespace for testing TCP traffic shifting and label it to enable automatic sidecar injection.

```
$ kubectl create namespace istio-io-tcp-traffic-shifting  
$ kubectl label namespace istio-io-tcp-traffic-shifting istio-injection=enabled
```

2. Deploy the sleep sample app to use as a test source for sending requests.

```
$ kubectl apply -f @samples/sleep/sleep.yaml@ -n istio-io-tcp-traffic-shifting
```

3. Deploy the v1 and v2 versions of the tcp-echo microservice.

```
$ kubectl apply -f @samples/tcp-echo/tcp-echo-services.yaml@ -n istio-io-tcp-traffic-shifting
```

4. Follow the instructions in Determining the ingress IP and ports to define the TCP\_INGRESS\_PORT and INGRESS\_HOST environment variables.

# Apply weight-based TCP routing

1. Route all TCP traffic to the v1 version of the `tcp-echo` microservice.

```
$ kubectl apply -f @samples/tcp-echo/tcp-echo-all-v1.yaml@  
-n istio-io-tcp-traffic-shifting
```

2. Confirm that the `tcp-echo` service is up and running by sending some TCP traffic from the `sleep` client.

```
$ for i in {1..20}; do \
  kubectl exec "$(kubectl get pod -l app=sleep -n istio-io-tc
  p-traffic-shifting -o jsonpath={.items..metadata.name})" \
  -c sleep -n istio-io-tcp-traffic-shifting -- sh -c "(date;
  sleep 1) | nc $INGRESS_HOST $TCP_INGRESS_PORT"; \
done

one Mon Nov 12 23:24:57 UTC 2018
one Mon Nov 12 23:25:00 UTC 2018
one Mon Nov 12 23:25:02 UTC 2018
one Mon Nov 12 23:25:05 UTC 2018
one Mon Nov 12 23:25:07 UTC 2018
one Mon Nov 12 23:25:10 UTC 2018
one Mon Nov 12 23:25:12 UTC 2018
one Mon Nov 12 23:25:15 UTC 2018
one Mon Nov 12 23:25:17 UTC 2018
one Mon Nov 12 23:25:19 UTC 2018
...

```

You should notice that all the timestamps have a

prefix of *one*, which means that all traffic was routed to the v1 version of the tcp-echo service.

3. Transfer 20% of the traffic from tcp-echo:v1 to tcp-echo:v2 with the following command:

```
$ kubectl apply -f @samples/tcp-echo/tcp-echo-20-v2.yaml@ -n istio-io-tcp-traffic-shifting
```

Wait a few seconds for the new rules to propagate.

4. Confirm that the rule was replaced:

```
$ kubectl get virtualservice tcp-echo -o yaml -n istio-io-tcp-traffic-shifting
apiVersion: networking.istio.io/v1beta1
```

```
kind: VirtualService
...
spec:
...
tcp:
- match:
  - port: 31400
    route:
    - destination:
        host: tcp-echo
        port:
          number: 9000
          subset: v1
        weight: 80
    - destination:
        host: tcp-echo
        port:
          number: 9000
          subset: v2
        weight: 20
```

5. Send some more TCP traffic to the `tcp-echo` microservice.

```
$ for i in {1..20}; do \
  kubectl exec "$(kubectl get pod -l app=sleep -n istio-io-tc
  p-traffic-shifting -o jsonpath={.items..metadata.name})" \
  -c sleep -n istio-io-tcp-traffic-shifting -- sh -c "(date;
  sleep 1) | nc $INGRESS_HOST $TCP_INGRESS_PORT"; \
done

one Mon Nov 12 23:38:45 UTC 2018
two Mon Nov 12 23:38:47 UTC 2018
one Mon Nov 12 23:38:50 UTC 2018
one Mon Nov 12 23:38:52 UTC 2018
one Mon Nov 12 23:38:55 UTC 2018
two Mon Nov 12 23:38:57 UTC 2018
one Mon Nov 12 23:39:00 UTC 2018
one Mon Nov 12 23:39:02 UTC 2018
one Mon Nov 12 23:39:05 UTC 2018
one Mon Nov 12 23:39:07 UTC 2018
...

```

You should now notice that about 20% of the

timestamps have a prefix of `two`, which means that 80% of the TCP traffic was routed to the `v1` version of the `tcp-echo` service, while 20% was routed to `v2`.

## Understanding what happened

In this task you partially migrated TCP traffic from an old to new version of the `tcp-echo` service using Istio's weighted routing feature. Note that this is very

different than doing version migration using the deployment features of container orchestration platforms, which use instance scaling to manage the traffic.

With Istio, you can allow the two versions of the `tcp-echo` service to scale up and down independently, without affecting the traffic distribution between them.

For more information about version routing with autoscaling, check out the blog article [Canary Deployments using Istio](#).

# Cleanup

1. Remove the sleep sample, tcp-echo application, and routing rules:

```
$ kubectl delete -f @samples/tcp-echo/tcp-echo-all-v1.yaml@  
-n istio-io-tcp-traffic-shifting  
$ kubectl delete -f @samples/tcp-echo/tcp-echo-services.yaml@ -n istio-io-tcp-traffic-shifting  
$ kubectl delete -f @samples/sleep/sleep.yaml@ -n istio-io-tcp-traffic-shifting  
$ kubectl delete namespace istio-io-tcp-traffic-shifting
```



# Request Timeouts

⌚ 3 minute read ✓ page test

---

This task shows you how to setup request timeouts in Envoy using Istio.

## Before you begin

- Setup Istio by following the instructions in the Installation guide.
- Deploy the Bookinfo sample application including the default destination rules.
- Initialize the application version routing by running the following command:

```
$ kubectl apply -f @samples/bookinfo/networking/virtual-service-all-v1.yaml@
```

## Request timeouts

A timeout for HTTP requests can be specified using the `timeout` field of the route rule. By default, the request timeout is disabled, but in this task you override the `reviews` service timeout to 1 second. To see its effect, however, you also introduce an artificial 2 second delay in calls to the `ratings` service.

1. Route requests to v2 of the `reviews` service, i.e., a version that calls the `ratings` service:

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
    - reviews
  http:
    - route:
        - destination:
            host: reviews
            subset: v2
EOF
```

2. Add a 2 second delay to calls to the ratings service:

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
  - ratings
  http:
  - fault:
      delay:
        percent: 100
        fixedDelay: 2s
      route:
      - destination:
          host: ratings
          subset: v1
EOF
```

### 3. Open the Bookinfo URL

`http://$GATEWAY_URL/productpage` in your browser.

You should see the Bookinfo application working normally (with ratings stars displayed), but there is a 2 second delay whenever you refresh the page.

### 4. Now add a half second request timeout for calls to the reviews service:

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v2
    timeout: 0.5s
EOF
```

## 5. Refresh the Bookinfo web page.

You should now see that it returns in about 1

second, instead of 2, and the reviews are unavailable.



The reason that the response takes 1 second, even though the timeout is configured at half a second, is because there is a hard-coded retry in the `productpage` service, so it calls the timing out `reviews` service twice before returning.

# Understanding what happened

In this task, you used Istio to set the request timeout for calls to the `reviews` microservice to half a second. By default the request timeout is disabled. Since the `reviews` service subsequently calls the `ratings` service when handling requests, you used Istio to inject a 2 second delay in calls to `ratings` to cause the `reviews` service to take longer than half a second to complete and consequently you could see the timeout in action.

You observed that instead of displaying reviews, the

Bookinfo product page (which calls the `reviews` service to populate the page) displayed the message: Sorry, product reviews are currently unavailable for this book. This was the result of it receiving the timeout error from the `reviews` service.

If you examine the fault injection task, you'll find out that the `productpage` microservice also has its own application-level timeout (3 seconds) for calls to the `reviews` microservice. Notice that in this task you used an Istio route rule to set the timeout to half a second. Had you instead set the timeout to something greater than 3 seconds (such as 4 seconds) the timeout would have had no effect since the more restrictive of the

two takes precedence. More details can be found [here](#).

One more thing to note about timeouts in Istio is that in addition to overriding them in route rules, as you did in this task, they can also be overridden on a per-request basis if the application adds an `x-envoy-upstream-rq-timeout-ms` header on outbound requests. In the header, the timeout is specified in milliseconds instead of seconds.

## Cleanup

- Remove the application routing rules:

```
$ kubectl delete -f @samples/bookinfo/networking/virtual-service-all-v1.yaml@
```

- If you are not planning to explore any follow-on tasks, see the Bookinfo cleanup instructions to shutdown the application.

# Circuit Breaking

⌚ 7 minute read ✓ page test

---

This task shows you how to configure circuit breaking for connections, requests, and outlier detection.

Circuit breaking is an important pattern for creating resilient microservice applications. Circuit breaking allows you to write applications that limit the impact of failures, latency spikes, and other undesirable

effects of network peculiarities.

In this task, you will configure circuit breaking rules and then test the configuration by intentionally “tripping” the circuit breaker.

## Before you begin

- Setup Istio by following the instructions in the Installation guide.
- Start the httpbin sample.

If you have enabled automatic sidecar injection,  
deploy the `httpbin` service:

```
$ kubectl apply -f @samples/httpbin/httpbin.yaml@
```

Otherwise, you have to manually inject the  
sidecar before deploying the `httpbin` application:

```
$ kubectl apply -f <(istioctl kube-inject -f @samples/httpb  
in/httpbin.yaml@)
```

The `httpbin` application serves as the backend service  
for this task.

# Configuring the circuit breaker

1. Create a destination rule to apply circuit breaking settings when calling the `httpbin` service:

If you installed/configured Istio with mutual TLS authentication enabled, you must add a TLS traffic policy mode:



`ISTIO_MUTUAL` to the `DestinationRule` before applying it. Otherwise requests will

generate 503 errors as described here.

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: httpbin
spec:
  host: httpbin
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 1
      http:
        http1MaxPendingRequests: 1
        maxRequestsPerConnection: 1
    outlierDetection:
      consecutive5xxErrors: 1
```

```
interval: 1s  
baseEjectionTime: 3m  
maxEjectionPercent: 100
```

EOF

2. Verify the destination rule was created correctly:

```
$ kubectl get destinationrule httpbin -o yaml
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
...
spec:
  host: httpbin
  trafficPolicy:
    connectionPool:
      http:
        http1MaxPendingRequests: 1
        maxRequestsPerConnection: 1
      tcp:
        maxConnections: 1
    outlierDetection:
      baseEjectionTime: 3m
      consecutive5xxErrors: 1
      interval: 1s
      maxEjectionPercent: 100
```

# Adding a client

Create a client to send traffic to the `httpbin` service. The client is a simple load-testing client called `fortio`. Fortio lets you control the number of connections, concurrency, and delays for outgoing HTTP calls. You will use this client to “trip” the circuit breaker policies you set in the `DestinationRule`.

1. Inject the client with the Istio sidecar proxy so network interactions are governed by Istio.

If you have enabled automatic sidecar injection, deploy the `fortio` service:

```
$ kubectl apply -f @samples/httpbin/sample-client/fortio-deploy.yaml@
```

Otherwise, you have to manually inject the sidecar before deploying the `fortio` application:

```
$ kubectl apply -f <(istioctl kube-inject -f @samples/httpbin/sample-client/fortio-deploy.yaml@)
```

2. Log in to the client pod and use the `fortio` tool to call `httpbin`. Pass in `curl` to indicate that you just want to make one call:

```
$ export FORTIO_POD=$(kubectl get pods -l app=fortio -o 'jsonpath={.items[0].metadata.name}')
$ kubectl exec "$FORTIO_POD" -c fortio -- /usr/bin/fortio curl -quiet http://httpbin:8000/get
```

HTTP/1.1 200 OK

server: envoy

date: Tue, 25 Feb 2020 20:25:52 GMT

content-type: application/json

content-length: 586

access-control-allow-origin: \*

access-control-allow-credentials: true

x-envoy-upstream-service-time: 36

{

    "args": {},

    "headers": {

        "Content-Length": "0",

        "Host": "httpbin:8000",

        "User-Agent": "fortio.org/fortio-1.3.1",

        "X-B3-Parentspanid": "8fc453fb1dec2c22",

        "X-B3-Sampled": "1",

        "X-B3-Spanid": "071d7f06bc94943c",

        "X-B3-Traceid": "86a929a0e76cda378fc453fb1dec2c22",

        "X-Forwarded-Client-Cert": "By=spiffe://cluster.local/n

```
s/default/sa/httpbin;Hash=68bbaedfe01ef4cb99e17358ff63e92d  
04a4ce831a35ab9a31d3c8e06adb038;Subject=\"\";URI=spiffe://c  
luster.local/ns/default/sa/default"  
},  
"origin": "127.0.0.1",  
"url": "http://httpbin:8000/get"  
}  
]
```

You can see the request succeeded! Now, it's time to break something.

# Tripping the circuit breaker

In the DestinationRule settings, you specified maxConnections: 1 and http1MaxPendingRequests: 1. These rules indicate that if you exceed more than one connection and request concurrently, you should see some failures when the `istio-proxy` opens the circuit for further requests and connections.

1. Call the service with two concurrent connections (-c 2) and send 20 requests (-n 20):

```
$ kubectl exec "$FORTIO_POD" -c fortio -- /usr/bin/fortio load -c 2 -qps 0 -n 20 -loglevel Warning http://httpbin:8000/get
20:33:46 I logger.go:97> Log level is now 3 Warning (was 2 Info)
Fortio 1.3.1 running at 0 queries per second, 6->6 procs, f
```

```
or 20 calls: http://httpbin:8000/get
Starting at max qps with 2 thread(s) [gomax 6] for exactly
20 calls (10 per thread + 0)
20:33:46 W http_client.go:679> Parsed non ok code 503 (HTTP
/1.1 503)
20:33:47 W http_client.go:679> Parsed non ok code 503 (HTTP
/1.1 503)
20:33:47 W http_client.go:679> Parsed non ok code 503 (HTTP
/1.1 503)
Ended after 59.8524ms : 20 calls. qps=334.16
Aggregated Function Time : count 20 avg 0.0056869 +/- 0.003
869 min 0.000499 max 0.0144329 sum 0.113738
# range, mid point, percentile, count
>= 0.000499 <= 0.001 , 0.0007495 , 10.00, 2
> 0.001 <= 0.002 , 0.0015 , 15.00, 1
> 0.003 <= 0.004 , 0.0035 , 45.00, 6
> 0.004 <= 0.005 , 0.0045 , 55.00, 2
> 0.005 <= 0.006 , 0.0055 , 60.00, 1
> 0.006 <= 0.007 , 0.0065 , 70.00, 2
> 0.007 <= 0.008 , 0.0075 , 80.00, 2
```

```
> 0.008 <= 0.009 , 0.0085 , 85.00, 1
> 0.011 <= 0.012 , 0.0115 , 90.00, 1
> 0.012 <= 0.014 , 0.013 , 95.00, 1
> 0.014 <= 0.0144329 , 0.0142165 , 100.00, 1
# target 50% 0.0045
# target 75% 0.0075
# target 90% 0.012
# target 99% 0.0143463
# target 99.9% 0.0144242
Sockets used: 4 (for perfect keepalive, would be 2)
Code 200 : 17 (85.0 %)
Code 503 : 3 (15.0 %)
Response Header Sizes : count 20 avg 195.65 +/- 82.19 min 0
max 231 sum 3913
Response Body/Total Sizes : count 20 avg 729.9 +/- 205.4 mi
n 241 max 817 sum 14598
All done 20 calls (plus 0 warmup) 5.687 ms avg, 334.2 qps
```

It's interesting to see that almost all requests

made it through! The `istio-proxy` does allow for some leeway.

```
Code 200 : 17 (85.0 %)
Code 503 : 3 (15.0 %)
```

2. Bring the number of concurrent connections up to 3:

```
$ kubectl exec "$FORTIO_POD" -c fortio -- /usr/bin/fortio load -c 3 -qps 0 -n 30 -loglevel Warning http://httpbin:8000/get
20:32:30 I logger.go:97> Log level is now 3 Warning (was 2 Info)
Fortio 1.3.1 running at 0 queries per second, 6->6 procs, for 30 calls: http://httpbin:8000/get
Starting at max qps with 3 thread(s) [gomax 6] for exactly 30 calls (10 per thread + 0)
```



```
20:32:30 W http_client.go:679> Parsed non ok code 503 (HTTP  
/1.1 503)  
20:32:30 W http_client.go:679> Parsed non ok code 503 (HTTP  
/1.1 503)  
20:32:30 W http_client.go:679> Parsed non ok code 503 (HTTP  
/1.1 503)  
20:32:30 W http_client.go:679> Parsed non ok code 503 (HTTP  
/1.1 503)  
20:32:30 W http_client.go:679> Parsed non ok code 503 (HTTP  
/1.1 503)  
20:32:30 W http_client.go:679> Parsed non ok code 503 (HTTP  
/1.1 503)  
20:32:30 W http_client.go:679> Parsed non ok code 503 (HTTP  
/1.1 503)  
20:32:30 W http_client.go:679> Parsed non ok code 503 (HTTP  
/1.1 503)  
20:32:30 W http_client.go:679> Parsed non ok code 503 (HTTP  
/1.1 503)  
20:32:30 W http_client.go:679> Parsed non ok code 503 (HTTP  
/1.1 503)  
20:32:30 W http_client.go:679> Parsed non ok code 503 (HTTP  
/1.1 503)  
20:32:30 W http_client.go:679> Parsed non ok code 503 (HTTP  
/1.1 503)  
Ended after 51.9946ms : 30 calls. qps=576.98  
Aggregated Function Time : count 30 avg 0.0040001633 +/- 0.
```

003447 min 0.0004298 max 0.015943 sum 0.1200049  
# range, mid point, percentile, count  
>= 0.0004298 <= 0.001 , 0.0007149 , 16.67, 5  
> 0.001 <= 0.002 , 0.0015 , 36.67, 6  
> 0.002 <= 0.003 , 0.0025 , 50.00, 4  
> 0.003 <= 0.004 , 0.0035 , 60.00, 3  
> 0.004 <= 0.005 , 0.0045 , 66.67, 2  
> 0.005 <= 0.006 , 0.0055 , 76.67, 3  
> 0.006 <= 0.007 , 0.0065 , 83.33, 2  
> 0.007 <= 0.008 , 0.0075 , 86.67, 1  
> 0.008 <= 0.009 , 0.0085 , 90.00, 1  
> 0.009 <= 0.01 , 0.0095 , 96.67, 2  
> 0.014 <= 0.015943 , 0.0149715 , 100.00, 1  
# target 50% 0.003  
# target 75% 0.00583333  
# target 90% 0.009  
# target 99% 0.0153601  
# target 99.9% 0.0158847  
Sockets used: 20 (for perfect keepalive, would be 3)  
Code 200 : 11 (36.7 %)

```
Code 503 : 19 (63.3 %)
Response Header Sizes : count 30 avg 84.366667 +/- 110.9 mi
n 0 max 231 sum 2531
Response Body/Total Sizes : count 30 avg 451.866667 +/- 277.
1 min 241 max 817 sum 13556
All done 30 calls (plus 0 warmup) 4.000 ms avg, 577.0 qps
```

Now you start to see the expected circuit breaking behavior. Only 36.7% of the requests succeeded and the rest were trapped by circuit breaking:

```
Code 200 : 11 (36.7 %)
Code 503 : 19 (63.3 %)
```

3. Query the istio-proxy stats to see more:

```
$ kubectl exec "$FORTIO_POD" -c istio-proxy -- pilot-agent
request GET stats | grep httpbin | grep pending
cluster.outbound|8000||httpbin.default.svc.cluster.local.ci
rcuit_breakers.default.remaining_pending: 1
cluster.outbound|8000||httpbin.default.svc.cluster.local.ci
rcuit_breakers.default.rq_pending_open: 0
cluster.outbound|8000||httpbin.default.svc.cluster.local.ci
rcuit_breakers.high.rq_pending_open: 0
cluster.outbound|8000||httpbin.default.svc.cluster.local.up
stream_rq_pending_active: 0
cluster.outbound|8000||httpbin.default.svc.cluster.local.up
stream_rq_pending_failure_eject: 0
cluster.outbound|8000||httpbin.default.svc.cluster.local.up
stream_rq_pending_overflow: 21
cluster.outbound|8000||httpbin.default.svc.cluster.local.up
stream_rq_pending_total: 29
```

You can see 21 for the upstream\_rq\_pending\_overflow value which means 21 calls so far have been

flagged for circuit breaking.

# Cleaning up

1. Remove the rules:

```
$ kubectl delete destinationrule httpbin
```

2. Shutdown the httpbin service and client:

```
$ kubectl delete deploy httpbin fortio-deploy
$ kubectl delete svc httpbin fortio
```

# Mirroring

⌚ 4 minute read ✓ page test

---

This task demonstrates the traffic mirroring capabilities of Istio.

Traffic mirroring, also called shadowing, is a powerful concept that allows feature teams to bring changes to production with as little risk as possible. Mirroring sends a copy of live traffic to a mirrored service. The

mirrored traffic happens out of band of the critical request path for the primary service.

In this task, you will first force all traffic to `v1` of a test service. Then, you will apply a rule to mirror a portion of traffic to `v2`.

## Before you begin

- Set up Istio by following the instructions in the Installation guide.

- Start by deploying two versions of the httpbin service that have access logging enabled:

## **httpbin-v1:**

```
$ cat <<EOF | istioctl kube-inject -f - | kubectl create -f  
-  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: httpbin-v1  
spec:  
  replicas: 1  
  selector:  
    matchLabels:  
      app: httpbin  
      version: v1  
  template:  
    metadata:
```

```
labels:
  app: httpbin
  version: v1
spec:
  containers:
    - image: docker.io/kennethreitz/httpbin
      imagePullPolicy: IfNotPresent
      name: httpbin
      command: ["gunicorn", "--access-logfile", "-", "-b"
, "0.0.0.0:80", "httpbin:app"]
      ports:
        - containerPort: 80
EOF
```

## httpbin-v2:

```
$ cat <<EOF | istioctl kube-inject -f - | kubectl create -f
-
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  name: httpbin-v2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: httpbin
      version: v2
  template:
    metadata:
      labels:
        app: httpbin
        version: v2
    spec:
      containers:
        - image: docker.io/kennethreitz/httpbin
          imagePullPolicy: IfNotPresent
          name: httpbin
          command: ["gunicorn", "--access-logfile", "-", "-b"
```

```
, "0.0.0.0:80", "httpbin:app"]  
    ports:  
        - containerPort: 80  
EOF
```

## **httpbin Kubernetes service:**

```
$ kubectl create -f - <<EOF
apiVersion: v1
kind: Service
metadata:
  name: httpbin
  labels:
    app: httpbin
spec:
  ports:
  - name: http
    port: 8000
    targetPort: 80
  selector:
    app: httpbin
EOF
```

- Start the sleep service so you can use curl to provide load:

## **sleep service:**

```
$ cat <<EOF | istioctl kube-inject -f - | kubectl create -f  
-  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: sleep  
spec:  
  replicas: 1  
  selector:  
    matchLabels:  
      app: sleep  
  template:  
    metadata:  
      labels:  
        app: sleep  
  spec:  
    containers:  
    - name: sleep
```

```
image: curlimages/curl
command: ["/bin/sleep", "3650d"]
imagePullPolicy: IfNotPresent
```

EOF

## Creating a default routing policy

By default Kubernetes load balances across both versions of the `httpbin` service. In this step, you will change that behavior so that all traffic goes to `v1`.

# 1. Create a default route rule to route all traffic to v1 of the service:

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: httpbin
spec:
  hosts:
    - httpbin
  http:
    - route:
        - destination:
            host: httpbin
            subset: v1
            weight: 100
---
apiVersion: networking.istio.io/v1alpha3
```

```
kind: DestinationRule
metadata:
  name: httpbin
spec:
  host: httpbin
  subsets:
    - name: v1
      labels:
        version: v1
    - name: v2
      labels:
        version: v2
EOF
```

Now all traffic goes to the `httpbin:v1` service.

## 2. Send some traffic to the service:

```
$ export SLEEP_POD=$(kubectl get pod -l app=sleep -o jsonpath={.items..metadata.name})
```

```
$ kubectl exec "${SLEEP_POD}" -c sleep -- curl -sS http://httpbin:8000/headers
{
  "headers": {
    "Accept": "*/*",
    "Content-Length": "0",
    "Host": "httpbin:8000",
    "User-Agent": "curl/7.35.0",
    "X-B3-Parentspanid": "57784f8bff90ae0b",
    "X-B3-Sampled": "1",
    "X-B3-Spanid": "3289ae7257c3f159",
    "X-B3-Traceid": "b56eebd279a76f0b57784f8bff90ae0b",
    "X-Envoy-Attempt-Count": "1",
    "X-Forwarded-Client-Cert": "By=spiffe://cluster.local/ns/default/sa/default;Hash=20afebed6da091c850264cc751b8c9306abac02993f80bdb76282237422bd098;Subject=\"\";URI=spiffe://cluster.local/ns/default/sa/default"
  }
}
```

3. Check the logs for v1 and v2 of the httpbin pods.  
You should see access log entries for v1 and none for v2:

```
$ export V1_POD=$(kubectl get pod -l app=httpbin,version=v1  
-o jsonpath={.items..metadata.name})  
$ kubectl logs "$V1_POD" -c httpbin  
127.0.0.1 - - [07/Mar/2018:19:02:43 +0000] "GET /headers HT  
TP/1.1" 200 321 "-" "curl/7.35.0"
```

```
$ export V2_POD=$(kubectl get pod -l app=httpbin,version=v2  
-o jsonpath={.items..metadata.name})  
$ kubectl logs "$V2_POD" -c httpbin  
<none>
```

# Mirroring traffic to v2

1. Change the route rule to mirror traffic to v2:

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: httpbin
spec:
  hosts:
    - httpbin
  http:
    - route:
        - destination:
            host: httpbin
            subset: v1
            weight: 100
        mirror:
            host: httpbin
            subset: v2
  mirrorPercentage:
    value: 100.0
EOF
```

This route rule sends 100% of the traffic to `v1`. The last stanza specifies that you want to mirror (i.e., also send) 100% of the same traffic to the `httpbin:v2` service. When traffic gets mirrored, the requests are sent to the mirrored service with their Host/Authority headers appended with `-shadow`. For example, `cluster-1` becomes `cluster-1-shadow`.

Also, it is important to note that these requests are mirrored as “fire and forget”, which means that the responses are discarded.

You can use the `value` field under the `mirrorPercentage` field to mirror a fraction of the

traffic, instead of mirroring all requests. If this field is absent, all traffic will be mirrored.

## 2. Send in traffic:

```
$ kubectl exec "${SLEEP_POD}" -c sleep -- curl -sS http://httpbin:8000/headers
```

Now, you should see access logging for both `v1` and `v2`. The access logs created in `v2` are the mirrored requests that are actually going to `v1`.

```
$ kubectl logs "$V1_POD" -c httpbin
127.0.0.1 - - [07/Mar/2018:19:02:43 +0000] "GET /headers HTTP/1.1" 200 321 "-" "curl/7.35.0"
127.0.0.1 - - [07/Mar/2018:19:26:44 +0000] "GET /headers HTTP/1.1" 200 321 "-" "curl/7.35.0"
```

```
$ kubectl logs "$V2_POD" -c httpbin  
127.0.0.1 - - [07/Mar/2018:19:26:44 +0000] "GET /headers HT  
TP/1.1" 200 361 "-" "curl/7.35.0"
```

# Cleaning up

1. Remove the rules:

```
$ kubectl delete virtualservice httpbin  
$ kubectl delete destinationrule httpbin
```

2. Shutdown the httpbin service and client:

```
$ kubectl delete deploy httpbin-v1 httpbin-v2 sleep  
$ kubectl delete svc httpbin
```

# Before you begin

⌚ 3 minute read ✓ page test

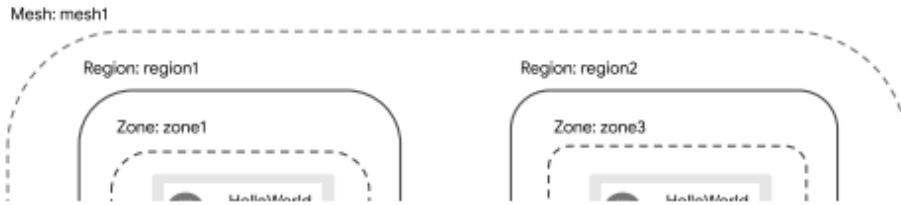
---

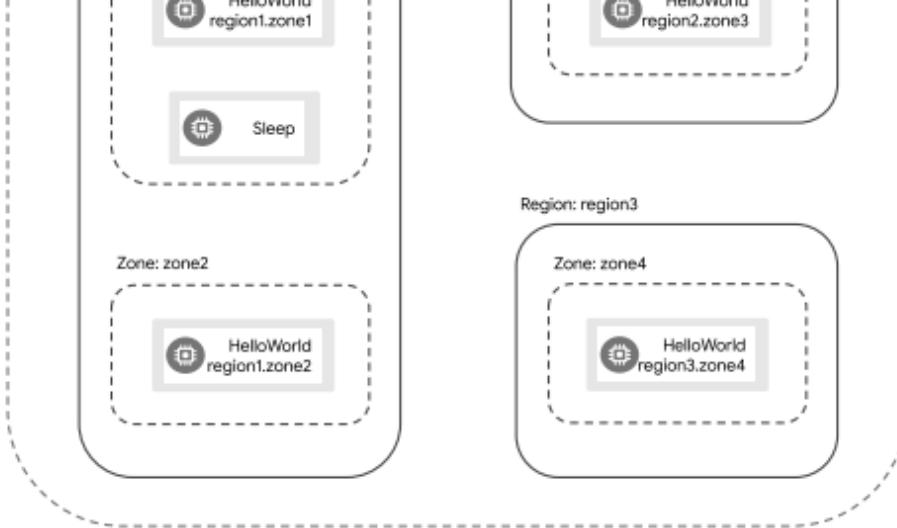
Before you begin the locality load balancing tasks, you must first install Istio on multiple clusters. The clusters must span three regions, containing four availability zones. The number of clusters required may vary based on the capabilities offered by your cloud provider.

For simplicity, we will assume that there is only a single primary cluster in the mesh.

This simplifies the process of configuring the control plane, since changes only need to be applied to one cluster.

We will deploy several instances of the `HelloWorld` application as follows:





Setup for locality load balancing  
tasks

# Environment Variables

This guide assumes that all clusters will be accessed through contexts in the default Kubernetes configuration file. The following environment variables will be used for the various contexts:

Variable	Description
CTX_PRIMARY	The context used for applying configuration to the primary cluster.
CTX_R1_Z1	The context used to interact with pods in region1.zone1.
CTX_R1_Z2	The context used to interact with pods in region1.zone2.

2	in region1.zone2.
CTX_R2_Z 3	The context used to interact with pods in region2.zone3.
CTX_R3_Z 4	The context used to interact with pods in region3.zone4.

## Create the sample namespace

To begin, generate yaml for the sample namespace

with automatic sidecar injection enabled:

```
$ cat <<EOF > sample.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: sample
  labels:
    istio-injection: enabled
EOF
```

Add the sample namespace to each cluster:

```
$ for CTX in "$CTX_PRIMARY" "$CTX_R1_Z1" "$CTX_R1_Z2" "$CTX_R2_Z3" "$CTX_R3_Z4"; \
do \
  kubectl --context="$CTX" apply -f sample.yaml; \
done
```

## Deploy HelloWorld

Generate the HelloWorld YAML for each locality, using the locality as the version string:

```
$ for LOC in "region1.zone1" "region1.zone2" "region2.zone3" "re  
gion3.zone4"; \  
do \  
./@samples/helloworld/gen-helloworld.sh@ \  
--version "$LOC" > "helloworld-${LOC}.yaml"; \  
done
```

Apply the HelloWorld YAML to the appropriate cluster for each locality:

```
$ kubectl apply --context="${CTX_R1_Z1}" -n sample \  
-f helloworld-region1.zone1.yaml
```

```
$ kubectl apply --context="${CTX_R1_Z2}" -n sample \  
-f helloworld-region1.zone2.yaml
```

```
$ kubectl apply --context="${CTX_R2_Z3}" -n sample \  
-f helloworld-region2.zone3.yaml
```

```
$ kubectl apply --context="${CTX_R3_Z4}" -n sample \  
-f helloworld-region3.zone4.yaml
```

# Deploy Sleep

Deploy the Sleep application to region1 zone1:

```
$ kubectl apply --context="${CTX_R1_Z1}" \  
-f @samples/sleep/sleep.yaml@ -n sample
```

# Wait for HelloWorld pods

Wait until the HelloWorld pods in each zone are Running:

```
$ kubectl get pod --context="${CTX_R1_Z1}" -n sample -l app="helloWorld" \
    -l version="region1.zone1"
NAME                               READY   STATUS    RESOURCES
TARTS     AGE
helloworld-region1.zone1-86f77cd7b-cpxhv   2/2     Running   0
                                         30s
```

```
$ kubectl get pod --context="${CTX_R1_Z2}" -n sample -l app="hel
loworld" \
  -l version="region1.zone2"
NAME                               READY   STATUS    RES
TARTS     AGE
helloworld-region1.zone2-86f77cd7b-cpxhv   2/2     Running   0
                                         30s
```

```
$ kubectl get pod --context="${CTX_R2_Z3}" -n sample -l app="hel
loworld" \
  -l version="region2.zone3"
NAME                               READY   STATUS    RES
TARTS     AGE
helloworld-region2.zone3-86f77cd7b-cpxhv   2/2     Running   0
                                         30s
```

```
$ kubectl get pod --context="${CTX_R3_Z4}" -n sample -l app="hel
loworld" \
    -l version="region3.zone4"
NAME                                READY   STATUS    RES
TARTS      AGE
helloworld-region3.zone4-86f77cd7b-cpxhv   2/2     Running   0
                                         30s
```

**Congratulations!** You successfully configured the system and are now ready to begin the locality load balancing tasks!

# Next steps

You can now configure one of the following load balancing options:

- Locality failover
- Locality weighted distribution



Only one of the load balancing options should be configured, as they are mutually exclusive. Attempting to configure both may lead to unexpected behavior.



# Locality failover

⌚ 4 minute read ✓ page test

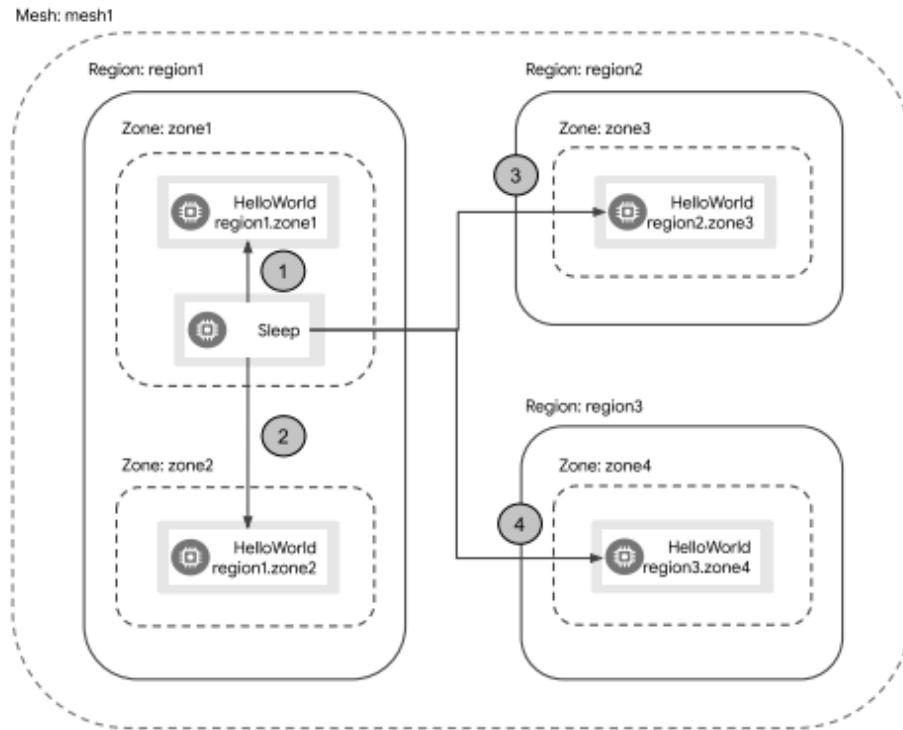
---

Follow this guide to configure your mesh for locality failover.

Before proceeding, be sure to complete the steps under before you begin.

In this task, you will use the `Sleep` pod in `region1.zone1` as the source of requests to the `HelloWorld` service.

You will then trigger failures that will cause failover between localities in the following sequence:



## Locality failover sequence

Internally, Envoy priorities are used to control failover. These priorities will be assigned as follows for traffic originating from the sleep pod (in region1 zone1):

Priority	Locality	Details
0	region1. zone1	Region, zone, and sub-zone all match.
1	None	Since this task doesn't use

		sub-zones, there are no matches for a different sub-zone.
2	region1. zone2	Different zone within the same region.
3	region2. zone3	No match, however failover is defined for region1->region2.
4	region3. zone4	No match and no failover defined for region1->region3.

# Configure locality failover

Apply a `DestinationRule` that configures the following:

- Outlier detection for the `HelloWorld` service. This is required in order for failover to function properly. In particular, it configures the sidecar proxies to know when endpoints for a service are unhealthy, eventually triggering a failover to the next locality.
- Failover policy between regions. This ensures that failover beyond a region boundary will behave predictably.

- Connection Pool policy that forces each HTTP request to use a new connection. This task utilizes Envoy's drain function to force a failover to the next locality. Once drained, Envoy will reject new connection requests. Since each request uses a new connection, this results in failover immediately following a drain. **This configuration is used for demonstration purposes only.**

```
$ kubectl --context="${CTX_PRIMARY}" apply -n sample -f - <<EOF
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: helloworld
```

```
spec:
```

```
  host: helloworld.sample.svc.cluster.local
  trafficPolicy:
    connectionPool:
      http:
        maxRequestsPerConnection: 1
  loadBalancer:
    simple: ROUND_ROBIN
  localityLbSetting:
    enabled: true
    failover:
      - from: region1
        to: region2
  outlierDetection:
    consecutive5xxErrors: 1
    interval: 1s
    baseEjectionTime: 1m
```

```
EOF
```

# Verify traffic stays in region1.zone1

Call the HelloWorld service from the sleep pod:

```
$ kubectl exec --context="${CTX_R1_Z1}" -n sample -c sleep \  
"$(kubectl get pod --context="${CTX_R1_Z1}" -n sample -l \  
app=sleep -o jsonpath='{.items[0].metadata.name}')"\ \  
-- curl -SSL helloworld.sample:5000/hello  
Hello version: region1.zone1, instance: helloworld-region1.zone1  
-86f77cd7b-cpxhv
```

Verify that the version in the response is region1.zone.

Repeat this several times and verify that the response is always the same.

## **Failover to** region1.zone2

Next, trigger a failover to region1.zone2. To do this, you drain the Envoy sidecar proxy **for** HelloWorld **in** region1.zone1:

```
$ kubectl --context="${CTX_R1_Z1}" exec \
"$(kubectl get pod --context="${CTX_R1_Z1}" -n sample -l app=helloworld \
-l version=region1.zone1 -o jsonpath='{.items[0].metadata.name}'")" \
-n sample -c istio-proxy -- curl -sSL -X POST 127.0.0.1:15000/
drain_listeners
```

Call the HelloWorld service from the sleep pod:

```
$ kubectl exec --context="${CTX_R1_Z1}" -n sample -c sleep \
"$(kubectl get pod --context="${CTX_R1_Z1}" -n sample -l \
app=sleep -o jsonpath='{.items[0].metadata.name}')"' \
-- curl -sSL helloworld.sample:5000/hello
Hello version: region1.zone2, instance: helloworld-region1.zone2
-86f77cd7b-cpxhv
```

The first call will fail, which triggers the failover.  
Repeat the command several more times and verify  
that the `version` in the response is always  
`region1.zone2`.

## **Failover to** `region2.zone3`

Now trigger a failover to `region2.zone3`. As you did previously, configure the `HelloWorld` in `region1.zone2` to fail when called:

```
$ kubectl --context="${CTX_R1_Z2}" exec \
"${(kubectl get pod --context="${CTX_R1_Z2}" -n sample -l app=helloworld \
-l version=region1.zone2 -o jsonpath='{.items[0].metadata.name}')}" \
-n sample -c istio-proxy -- curl -sSL -X POST 127.0.0.1:15000/
drain_listeners
```

Call the HelloWorld service from the sleep pod:

```
$ kubectl exec --context="${CTX_R1_Z1}" -n sample -c sleep \
"${(kubectl get pod --context="${CTX_R1_Z1}" -n sample -l \
app=sleep -o jsonpath='{.items[0].metadata.name}')}" \
-- curl -sSL helloworld.sample:5000/hello
Hello version: region2.zone3, instance: helloworld-region2.zone3
-86f77cd7b-cpxhv
```

The first call will fail, which triggers the failover.  
Repeat the command several more times and verify  
that the `version` in the response is always  
`region2.zone3`.

## **Failover to** `region3.zone4`

Now trigger a failover to `region3.zone4`. As you did previously, configure the `HelloWorld` in `region2.zone3` to fail when called:

```
$ kubectl --context="${CTX_R2_Z3}" exec \
"${(kubectl get pod --context="${CTX_R2_Z3}" -n sample -l app=helloworld \
-l version=region2.zone3 -o jsonpath='{.items[0].metadata.name}')}" \
-n sample -c istio-proxy -- curl -sSL -X POST 127.0.0.1:15000/
drain_listeners
```

Call the HelloWorld service from the sleep pod:

```
$ kubectl exec --context="${CTX_R1_Z1}" -n sample -c sleep \
"${(kubectl get pod --context="${CTX_R1_Z1}" -n sample -l \
app=sleep -o jsonpath='{.items[0].metadata.name}')}" \
-- curl -sSL helloworld.sample:5000/hello
Hello version: region3.zone4, instance: helloworld-region3.zone4
-86f77cd7b-cpxhv
```

The first call will fail, which triggers the failover.  
Repeat the command several more times and verify  
that the `version` in the response is always  
`region3.zone4`.

**Congratulations!** You successfully configured  
locality failover!

## Next steps

Cleanup resources and files from this task.



# Locality weighted distribution

⌚ 2 minute read ✓ page test

---

Follow this guide to configure the distribution of traffic across localities.

Before proceeding, be sure to complete the steps under before you begin.

In this task, you will use the Sleep pod in region1 zone1 as the source of requests to the HelloWorld service. You will configure Istio with the following distribution across localities:

Region	Zone	% of traffic
region1	zone1	70
region1	zone2	20
region2	zone3	0
region3	zone4	10

# Configure Weighted Distribution

Apply a `DestinationRule` that configures the following:

- Outlier detection for the `HelloWorld` service. This is required in order for distribution to function properly. In particular, it configures the sidecar proxies to know when endpoints for a service are unhealthy.

- Weighted Distribution for the HelloWorld service as described in the table above.

```
$ kubectl --context="${CTX_PRIMARY}" apply -n sample -f - <<EOF
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: helloworld
spec:
  host: helloworld.sample.svc.cluster.local
  trafficPolicy:
    loadBalancer:
      localityLbSetting:
        enabled: true
        distribute:
          - from: region1/zone1/*
            to:
              "region1/zone1/*": 70
              "region1/zone2/*": 20
```

```
    "region3/zone4/*": 10
outlierDetection:
  consecutive5xxErrors: 100
  interval: 1s
  baseEjectionTime: 1m
```

EOF

# Verify the distribution

Call the HelloWorld service from the Sleep pod:

```
$ kubectl exec --context="${CTX_R1_Z1}" -n sample -c sleep \  
"$(kubectl get pod --context="${CTX_R1_Z1}" -n sample -l \  
app=sleep -o jsonpath='{.items[0].metadata.name}')"\ \  
-- curl -sSL helloworld.sample:5000/hello
```

Repeat this a number of times and verify that the number of replies for each pod match the expected percentage in the table at the top of this guide.

**Congratulations!** You successfully configured locality distribution!

# Next steps

Cleanup resources and files from this task.

# Cleanup

---

Now that you've completed the locality load balancing tasks, let's cleanup.

## Remove generated files

```
$ rm -f sample.yaml helloworld-region*.zone*.yaml
```

# Remove the sample namespace

```
$ for CTX in "$CTX_PRIMARY" "$CTX_R1_Z1" "$CTX_R1_Z2" "$CTX_R2_Z  
3" "$CTX_R3_Z4"; \  
do \  
    kubectl --context="$CTX" delete ns sample --ignore-not-found  
=true; \  
done
```

**Congratulations!** You successfully completed the locality load balancing task!

# Ingress Gateways

⌚ 7 minute read ✓ page test

---

Along with support for Kubernetes Ingress, Istio offers another configuration model, Istio Gateway. A Gateway provides more extensive customization and flexibility than Ingress, and allows Istio features such as monitoring and route rules to be applied to traffic entering the cluster.

This task describes how to configure Istio to expose a service outside of the service mesh using an Istio Gateway.

## Before you begin

- Setup Istio by following the instructions in the Installation guide.
- Make sure your current directory is the `istio` directory.

- Start the httpbin sample.

If you have enabled automatic sidecar injection, deploy the httpbin service:

```
$ kubectl apply -f @samples/httpbin/httpbin.yaml@
```

Otherwise, you have to manually inject the sidecar before deploying the httpbin application:

```
$ kubectl apply -f <(istioctl kube-inject -f @samples/httpbin/httpbin.yaml@)
```

- Determine the ingress IP and ports as described in the following subsection.

# Determining the ingress IP and ports

Execute the following command to determine if your Kubernetes cluster is running in an environment that supports external load balancers:

```
$ kubectl get svc istio-ingressgateway -n istio-system
NAME                  TYPE        CLUSTER-IP      EXTERNAL-
IP      PORT(S)      AGE
istio-ingressgateway   LoadBalancer  172.21.109.129  130.211.1
0.121    ...        17h
```

If the EXTERNAL-IP value is set, your environment has

an external load balancer that you can use for the ingress gateway. If the EXTERNAL-IP value is <none> (or perpetually <pending>), your environment does not provide an external load balancer for the ingress gateway. In this case, you can access the gateway using the service's node port.

Choose the instructions corresponding to your environment:

**external load balancer**

node port

Follow these instructions if you have determined that your environment has an

external load balancer.

Set the ingress IP and ports:

```
$ export INGRESS_HOST=$(kubectl -n istio-system get service istio-ingressgateway -o jsonpath='{.status.loadBalancer.ingress[0].ip}')
$ export INGRESS_PORT=$(kubectl -n istio-system get service istio-ingressgateway -o jsonpath='{.spec.ports[?(@.name=="http2")].port}')
$ export SECURE_INGRESS_PORT=$(kubectl -n istio-system get service istio-ingressgateway -o jsonpath='{.spec.ports[?(@.name=="https")].port}')
$ export TCP_INGRESS_PORT=$(kubectl -n istio-system get service istio-ingressgateway -o jsonpath='{.spec.ports[?(@.name=="tcp")].port}')
```

In certain environments, the load balancer may be exposed using a host name, instead of an IP address. In this case, the ingress gateway's EXTERNAL-IP value will not be an IP address, but rather a host name, and the above command will have failed to set the INGRESS\_HOST environment variable. Use the following command to correct the INGRESS\_HOST value:



```
$ export INGRESS_HOST=$(kubectl -n istio-system get service istio-ingressgateway -o jsonpath='{.status.loadBalancer.ingress[0].hostname}')
```

# Configuring ingress using an Istio gateway

An ingress Gateway describes a load balancer

operating at the edge of the mesh that receives incoming HTTP/TCP connections. It configures exposed ports, protocols, etc. but, unlike Kubernetes Ingress Resources, does not include any traffic routing configuration. Traffic routing for ingress traffic is instead configured using Istio routing rules, exactly in the same way as for internal service requests.

Let's see how you can configure a [Gateway](#) on port 80 for HTTP traffic.

1. Create an Istio [Gateway](#):

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: httpbin-gateway
spec:
  selector:
    istio: ingressgateway # use Istio default gateway implementation
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "httpbin.example.com"
EOF
```

## 2. Configure routes for traffic entering via the

## Gateway:

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: httpbin
spec:
  hosts:
    - "httpbin.example.com"
  gateways:
    - httpbin-gateway
  http:
    - match:
        - uri:
            prefix: /status
        - uri:
            prefix: /delay
      route:
        - destination:
```

```
port:  
    number: 8000  
    host: httpbin  
EOF
```

You have now created a virtual service configuration for the `httpbin` service containing two route rules that allow traffic for paths `/status` and `/delay`.

The `gateways` list specifies that only requests through your `httpbin-gateway` are allowed. All other external requests will be rejected with a 404 response.



Internal requests from other services in the mesh are not subject to these rules but instead will default to round-robin routing. To apply these rules to internal calls as well, you can add the special value `mesh` to the list of gateways. Since the internal hostname for the service is probably different (e.g., `httpbin.default.svc.cluster.local`) from the external one, you will also need to add it to the `hosts` list. Refer to the operations guide [for more details](#).

### 3. Access the *httpbin* service using *curl*:

```
$ curl -s -I -HHost:httpbin.example.com "http://$INGRESS_HO  
ST:$INGRESS_PORT/status/200"  
HTTP/1.1 200 OK  
server: istio-envoy  
...
```

Note that you use the `-H` flag to set the *Host* HTTP header to “`httpbin.example.com`”. This is needed because your ingress `Gateway` is configured to handle “`httpbin.example.com`”, but in your test environment you have no DNS binding for that host and are simply sending your request to the

ingress IP.

4. Access any other URL that has not been explicitly exposed. You should see an HTTP 404 error:

```
$ curl -s -I -HHost:httpbin.example.com "http://$INGRESS_HO  
ST:$INGRESS_PORT/headers"  
HTTP/1.1 404 Not Found  
...
```

## Accessing ingress services using a browser

Entering the `httpbin` service URL in a browser won't work because you can't pass the `Host` header to a browser like you did with `curl`. In a real world situation, this is not a problem because you configure the requested host properly and DNS resolvable. Thus, you use the host's domain name in the URL, for example, `https://httpbin.example.com/status/200`.

To work around this problem for simple tests and demos, use a wildcard `*` value for the host in the Gateway and VirtualService configurations. For example, if you change your ingress configuration to the following:

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: httpbin-gateway
spec:
  selector:
    istio: ingressgateway # use Istio default gateway implementation
  servers:
    - port:
        number: 80
        name: http
        protocol: HTTP
      hosts:
        - "*"
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
```

```
name: httpbin
spec:
  hosts:
    - "*"
  gateways:
    - httpbin-gateway
  http:
    - match:
        - uri:
            prefix: /headers
        route:
          - destination:
              port:
                number: 8000
                host: httpbin
```

EOF

You can then use \$INGRESS\_HOST:\$INGRESS\_PORT in the browser URL. For example,

`http://$INGRESS_HOST:$INGRESS_PORT/headers` will display all the headers that your browser sends.

## Understanding what happened

The `Gateway` configuration resources allow external traffic to enter the Istio service mesh and make the traffic management and policy features of Istio available for edge services.

In the preceding steps, you created a service inside the service mesh and exposed an HTTP endpoint of the service to external traffic.

## Troubleshooting

1. Inspect the values of the `INGRESS_HOST` and `INGRESS_PORT` environment variables. Make sure they have valid values, according to the output of the following commands:

```
$ kubectl get svc -n istio-system  
$ echo "INGRESS_HOST=$INGRESS_HOST, INGRESS_PORT=$INGRESS_P  
ORT"
```

2. Check that you have no other Istio ingress gateways defined on the same port:

```
$ kubectl get gateway --all-namespaces
```

3. Check that you have no Kubernetes Ingress resources defined on the same IP and port:

```
$ kubectl get ingress --all-namespaces
```

4. If you have an external load balancer and it does not work for you, try to access the gateway using its

node port.

# Cleanup

Delete the Gateway and VirtualService configuration, and shutdown the httpbin service:

```
$ kubectl delete gateway httpbin-gateway
$ kubectl delete virtualservice httpbin
$ kubectl delete --ignore-not-found=true -f @samples/httpbin/httpbin.yaml@
```



# Secure Gateways

⌚ 9 minute read ✓ page test

---

The Control Ingress Traffic task describes how to configure an ingress gateway to expose an HTTP service to external traffic. This task shows how to expose a secure HTTPS service using either simple or mutual TLS.

# Before you begin

1. Perform the steps in the `Before you begin.` and `Determining the ingress IP and ports` sections of the `Control Ingress Traffic` task. After performing those steps you should have Istio and the `httpbin` service deployed, and the environment variables `INGRESS_HOST` and `SECURE_INGRESS_PORT` set.
2. For macOS users, verify that you use `curl` compiled with the LibreSSL library:

```
$ curl --version | grep LibreSSL  
curl 7.54.0 (x86_64-apple-darwin17.0) libcurl/7.54.0 LibreS  
SL/2.0.20 zlib/1.2.11 nghttp2/1.24.0
```

If the previous command outputs a version of LibreSSL as shown, your `curl` command should work correctly with the instructions in this task. Otherwise, try a different implementation of `curl`, for example on a Linux machine.

# Generate client and server certificates and keys

For this task you can use your favorite tool to generate certificates and keys. The commands below use openssl

1. Create a root certificate and private key to sign the certificates for your services:

```
$ openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 -subj '/O=example Inc./CN=example.com' -keyout example.com.key -out example.com.crt
```

2. Create a certificate and a private key for httpbin.example.com:

```
$ openssl req -out httpbin.example.com.csr -newkey rsa:2048  
-nodes -keyout httpbin.example.com.key -subj "/CN=httpbin.  
example.com/O=httpbin organization"  
$ openssl x509 -req -days 365 -CA example.com.crt -CAkey ex  
ample.com.key -set_serial 0 -in httpbin.example.com.csr -ou  
t httpbin.example.com.crt
```

# Configure a TLS ingress gateway for a single host

1. Ensure you have deployed the httpbin service from Before you begin.
2. Create a secret for the ingress gateway:

```
$ kubectl create -n istio-system secret tls httpbin-credential --key=httpbin.example.com.key --cert=httpbin.example.com.crt
```

3. Define a gateway with a servers: section for port 443, and specify values for credentialName to be httpbin-credential. The values are the same as the secret's name. The TLS mode should have the value of SIMPLE.

```
$ cat <<EOF | kubectl apply -f -
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: mygateway
spec:
  selector:
```

```
istio: ingressgateway # use istio default ingress gateway
servers:
- port:
    number: 443
    name: https
    protocol: HTTPS
  tls:
    mode: SIMPLE
    credentialName: httpbin-credential # must be the same
as secret
  hosts:
- httpbin.example.com
EOF
```

4. Configure the gateway's ingress traffic routes.  
Define the corresponding virtual service.

```
$ cat <<EOF | kubectl apply -f -
```

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: httpbin
spec:
  hosts:
    - "httpbin.example.com"
  gateways:
    - mygateway
  http:
    - match:
        - uri:
            prefix: /status
        - uri:
            prefix: /delay
      route:
        - destination:
            port:
              number: 8000
            host: httpbin
```

5. Send an HTTPS request to access the httpbin service through HTTPS:

```
$ curl -v -HHost:httpbin.example.com --resolve "httpbin.example.com:$SECURE_INGRESS_PORT:$INGRESS_HOST" \
--cacert example.com.crt "https://httpbin.example.com:$SECURE_INGRESS_PORT/status/418"
```

The httpbin service will return the 418 I'm a Teapot code.

6. Delete the gateway's secret and create a new one to change the ingress gateway's credentials.

```
$ kubectl -n istio-system delete secret httpbin-credential
```

```
$ mkdir new_certificates
$ openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 -subj '/O=example Inc./CN=example.com' -keyout new_certificates/example.com.key -out new_certificates/example.com.crt
$ openssl req -out new_certificates/httpbin.example.com.csr -newkey rsa:2048 -nodes -keyout new_certificates/httpbin.example.com.key -subj "/CN=httpbin.example.com/O=httpbin organization"
$ openssl x509 -req -days 365 -CA new_certificates/example.com.crt -CAkey new_certificates/example.com.key -set_serial 0 -in new_certificates/httpbin.example.com.csr -out new_certificates/httpbin.example.com.crt
$ kubectl create -n istio-system secret tls httpbin-credential \
--key=new_certificates/httpbin.example.com.key \
--cert=new_certificates/httpbin.example.com.crt
```

7. Access the httpbin service using curl using the

## new certificate chain:

```
$ curl -v -HHost:httpbin.example.com --resolve "httpbin.example.com:$SECURE_INGRESS_PORT:$INGRESS_HOST" \
--cacert new_certificates/example.com.crt "https://httpbin.example.com:$SECURE_INGRESS_PORT/status/418"

...
HTTP/2 418
...
-=[ teapot ]=
```



8. If you try to access `httpbin` with the previous certificate chain, the attempt now fails.

```
$ curl -v -HHost:httpbin.example.com --resolve "httpbin.example.com:$SECURE_INGRESS_PORT:$INGRESS_HOST" \
--cacert example.com.crt "https://httpbin.example.com:$SECURE_INGRESS_PORT/status/418"
...
* TLSv1.2 (OUT), TLS handshake, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Server hello (2):
* TLSv1.2 (IN), TLS handshake, Certificate (11):
* TLSv1.2 (OUT), TLS alert, Server hello (2):
* curl: (35) error:04FFF06A:rsa routines:CRYPTO_internal:block type is not 01
```

# Configure a TLS ingress

# gateway for multiple hosts

You can configure an ingress gateway for multiple hosts, `httpbin.example.com` and `helloworld-v1.example.com`, for example. The ingress gateway retrieves unique credentials corresponding to a specific `credentialName`.

1. To restore the credentials for `httpbin`, delete its secret and create it again.

```
$ kubectl -n istio-system delete secret httpbin-credential  
$ kubectl create -n istio-system secret tls httpbin-credential \  
--key=httpbin.example.com.key \  
--cert=httpbin.example.com.crt
```

## 2. Start the helloworld-v1 sample

```
$ cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Service
metadata:
  name: helloworld-v1
  labels:
    app: helloworld-v1
spec:
  ports:
  - name: http
    port: 5000
  selector:
    app: helloworld-v1
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: helloworld-v1
```

```
spec:
  replicas: 1
  selector:
    matchLabels:
      app: helloworld-v1
      version: v1
  template:
    metadata:
      labels:
        app: helloworld-v1
        version: v1
  spec:
    containers:
      - name: helloworld
        image: istio/examples-helloworld-v1
    resources:
      requests:
        cpu: "100m"
    imagePullPolicy: IfNotPresent #Always
  ports:
```

```
- containerPort: 5000  
EOF
```

### 3. Generate a certificate and a private key for helloworld-v1.example.com:

```
$ openssl req -out helloworld-v1.example.com.csr -newkey rsa:2048 -nodes -keyout helloworld-v1.example.com.key -subj "/CN=helloworld-v1.example.com/O=helloworld organization"  
$ openssl x509 -req -days 365 -CA example.com.crt -CAkey example.com.key -set_serial 1 -in helloworld-v1.example.com.csr -out helloworld-v1.example.com.crt
```

### 4. Create the helloworld-credential secret:

```
$ kubectl create -n istio-system secret tls helloworld-credential --key=helloworld-v1.example.com.key --cert=helloworld-v1.example.com.crt
```

5. Define a gateway with two server sections for port 443. Set the value of credentialName on each port to httpbin-credential and helloworld-credential respectively. Set TLS mode to SIMPLE.

```
$ cat <<EOF | kubectl apply -f -
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: mygateway
spec:
  selector:
    istio: ingressgateway # use istio default ingress gateway
  servers:
  - port:
      number: 443
      name: https-httpbin
```

```
    protocol: HTTPS
  tls:
    mode: SIMPLE
    credentialName: httpbin-credential
  hosts:
    - httpbin.example.com
  - port:
      number: 443
      name: https-helloworld
      protocol: HTTPS
    tls:
      mode: SIMPLE
      credentialName: helloworld-credential
    hosts:
      - helloworld-v1.example.com
EOF
```

6. Configure the gateway's traffic routes. Define the corresponding virtual service.

```
$ cat <<EOF | kubectl apply -f -
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: helloworld-v1
spec:
  hosts:
    - helloworld-v1.example.com
  gateways:
    - mygateway
  http:
    - match:
        - uri:
            exact: /hello
      route:
        - destination:
            host: helloworld-v1
            port:
              number: 5000
EOF
```

7. Send an HTTPS request to helloworld-v1.example.com:

```
$ curl -v -HHost:helloworld-v1.example.com --resolve "helloworld-v1.example.com:$SECURE_INGRESS_PORT:$INGRESS_HOST" \
--cacert example.com.crt "https://helloworld-v1.example.com
:$SECURE_INGRESS_PORT/hello"
HTTP/2 200
```

8. Send an HTTPS request to httpbin.example.com and still get a teapot in return:

```
$ curl -v -HHost:httpbin.example.com --resolve "httpbin.example.com:$SECURE_INGRESS_PORT:$INGRESS_HOST" \
--cacert example.com.crt "https://httpbin.example.com:$SECURE_INGRESS_PORT/status/418"
...
-[ teapot ]-
```



# Configure a mutual TLS

# ingress gateway

You can extend your gateway's definition to support mutual TLS. Change the credentials of the ingress gateway by deleting its secret and creating a new one. The server uses the CA certificate to verify its clients, and we must use the name `cacert` to hold the CA certificate.

```
$ kubectl -n istio-system delete secret httpbin-credential  
$ kubectl create -n istio-system secret generic httpbin-credential --from-file=tls.key=httpbin.example.com.key \  
--from-file=tls.crt=httpbin.example.com.crt --from-file=ca.crt=example.com.crt
```

# 1. Change the gateway's definition to set the TLS mode to MUTUAL.

```
$ cat <<EOF | kubectl apply -f -
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: mygateway
spec:
  selector:
    istio: ingressgateway # use istio default ingress gateway
  servers:
  - port:
      number: 443
      name: https
      protocol: HTTPS
    tls:
      mode: MUTUAL
```

```
credentialName: httpbin-credential # must be the same
as secret
hosts:
- httpbin.example.com
EOF
```

2. Attempt to send an HTTPS request using the prior approach and see how it fails:

```
$ curl -v -HHost:httpbin.example.com --resolve "httpbin.example.com:$SECURE_INGRESS_PORT:$INGRESS_HOST" \
--cacert example.com.crt "https://httpbin.example.com:$SECURE_INGRESS_PORT/status/418"
* TLSv1.3 (OUT), TLS handshake, Client hello (1):
* TLSv1.3 (IN), TLS handshake, Server hello (2):
* TLSv1.3 (IN), TLS handshake, Encrypted Extensions (8):
* TLSv1.3 (IN), TLS handshake, Request CERT (13):
* TLSv1.3 (IN), TLS handshake, Certificate (11):
* TLSv1.3 (IN), TLS handshake, CERT verify (15):
* TLSv1.3 (IN), TLS handshake, Finished (20):
* TLSv1.3 (OUT), TLS change cipher, Change cipher spec (1):
* TLSv1.3 (OUT), TLS handshake, Certificate (11):
* TLSv1.3 (OUT), TLS handshake, Finished (20):
* TLSv1.3 (IN), TLS alert, unknown (628):
* OpenSSL SSL_read: error:1409445C:SSL routines:ssl3_read_bytes:tlsv13 alert certificate required, errno 0
```

### 3. Generate client certificate and private key:

```
$ openssl req -out client.example.com.csr -newkey rsa:2048  
-nodes -keyout client.example.com.key -subj "/CN=client.exa  
mple.com/O=client organization"  
$ openssl x509 -req -days 365 -CA example.com.crt -CAkey ex  
ample.com.key -set_serial 1 -in client.example.com.csr -out  
client.example.com.crt
```

4. Pass a client certificate and private key to curl and resend the request. Pass your client's certificate with the --cert flag and your private key with the --key flag to curl.

```
$ curl -v -HHost:httpbin.example.com --resolve "httpbin.example.com:$SECURE_INGRESS_PORT:$INGRESS_HOST" \
--cacert example.com.crt --cert client.example.com.crt --key client.example.com.key \
"https://httpbin.example.com:$SECURE_INGRESS_PORT/status/418"
...
-= [ teapot ] =-
```



# More info

## Key formats

Istio supports reading a few different Secret formats, to support integration with various tools such as cert-manager:

- A TLS Secret with keys `tls.key` and `tls.crt`, as described above. For mutual TLS, a `ca.crt` key can be used.

- A generic Secret with keys `key` and `cert`. For mutual TLS, a `cacert` key can be used.
- A generic Secret with keys `key` and `cert`. For mutual TLS, a separate generic Secret named `<secret>-cacert`, with a `cacert` key. For example, `httpbin-credential` has `key` and `cert`, and `httpbin-credential-cacert` has `cacert`.
- The `cacert` key value can be a CA bundle consisting of concatenated individual CA certificates.

## SNI Routing

An HTTPS Gateway with a `hosts` field value other than `*` will perform SNI matching before forwarding a request, which may cause some requests to fail. See [configuring SNI routing](#) for details.

## Troubleshooting

- Inspect the values of the `INGRESS_HOST` and `SECURE_INGRESS_PORT` environment variables. Make sure they have valid values, according to the output of the following commands:

```
$ kubectl get svc -n istio-system
$ echo "INGRESS_HOST=$INGRESS_HOST, SECURE_INGRESS_PORT=$SE
CURE_INGRESS_PORT"
```

- Check the log of the `istio-ingressgateway` controller for error messages:

```
$ kubectl logs -n istio-system "$(
    kubectl get pod -l istio=
    ingressgateway \
    -n istio-system -o jsonpath='{.items[0].metadata.name}'")"
```

- If using macOS, verify you are using curl compiled with the LibreSSL library, as described in the Before you begin section.
- Verify that the secrets are successfully created in

## the `istio-system` namespace:

```
$ kubectl -n istio-system get secrets
```

`httpbin-credential` and `helloworld-credential` should show in the secrets list.

- Check the logs to verify that the ingress gateway agent has pushed the key/certificate pair to the ingress gateway.

```
$ kubectl logs -n istio-system "$(
    kubectl get pod -l istio=ingressgateway \
    -n istio-system -o jsonpath='{.items[0].metadata.name}')
```

The log should show that the `httpbin-credential`

secret was added. If using mutual TLS, then the httpbin-credential-cacert secret should also appear. Verify the log shows that the gateway agent receives SDS requests from the ingress gateway, that the resource's name is httpbin-credential, and that the ingress gateway obtained the key/certificate pair. If using mutual TLS, the log should show key/certificate was sent to the ingress gateway, that the gateway agent received the SDS request with the httpbin-credential-cacert resource name, and that the ingress gateway obtained the root certificate.

# Cleanup

1. Delete the gateway configuration, the virtual service definition, and the secrets:

```
$ kubectl delete gateway mygateway
$ kubectl delete virtualservice httpbin
$ kubectl delete --ignore-not-found=true -n istio-system secret httpbin-credential \
cret helloworld-credential
$ kubectl delete --ignore-not-found=true virtualservice hel
leworld-v1
```

2. Delete the certificates and keys:

```
$ rm -rf example.com.crt example.com.key httpbin.example.co  
m.crt httpbin.example.com.key httpbin.example.com.csr hello  
world-v1.example.com.crt helloworld-v1.example.com.key hell  
oworld-v1.example.com.csr client.example.com.crt client.exa  
mple.com.csr client.example.com.key ./new_certificates
```

### 3. Shutdown the httpbin and helloworld-v1 services:

```
$ kubectl delete deployment --ignore-not-found=true httpbin  
helloworld-v1  
$ kubectl delete service --ignore-not-found=true httpbin he  
lloworld-v1
```

# Ingress Gateway without TLS Termination

⌚ 4 minute read ✓ page test

---

---

The Securing Gateways with HTTPS task describes how to configure HTTPS ingress access to an HTTP service. This example describes how to configure HTTPS ingress access to an HTTPS service, i.e., configure an

ingress gateway to perform SNI passthrough, instead of TLS termination on incoming requests.

The example HTTPS service used for this task is a simple NGINX server. In the following steps you first deploy the NGINX service in your Kubernetes cluster. Then you configure a gateway to provide ingress access to the service via host `nginx.example.com`.

## **Generate client and server certificates and keys**

For this task you can use your favorite tool to generate certificates and keys. The commands below use openssl

1. Create a root certificate and private key to sign the certificate for your services:

```
$ openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 -subj '/O=example Inc./CN=example.com' -keyout example.com.key -out example.com.crt
```

2. Create a certificate and a private key for nginx.example.com:

```
$ openssl req -out nginx.example.com.csr -newkey rsa:2048 -nodes -keyout nginx.example.com.key -subj "/CN=nginx.example.com/O=some organization"
$ openssl x509 -req -sha256 -days 365 -CA example.com.crt -CAkey example.com.key -set_serial 0 -in nginx.example.com.csr -out nginx.example.com.crt
```

# Deploy an NGINX server

1. Create a Kubernetes Secret to hold the server's certificate.

```
$ kubectl create secret tls nginx-server-certs --key nginx.example.com.key --cert nginx.example.com.crt
```

## 2. Create a configuration file for the NGINX server:

```
$ cat <<\EOF > ./nginx.conf
events {
}

http {
    log_format main '$remote_addr - $remote_user [$time_local]
] $status '
    '"$request" $body_bytes_sent "$http_referer" '
    '"$http_user_agent" "$http_x_forwarded_for"';
    access_log /var/log/nginx/access.log main;
    error_log  /var/log/nginx/error.log;

server {
    listen 443 ssl;
```

```
root /usr/share/nginx/html;
index index.html;

server_name nginx.example.com;
ssl_certificate /etc/nginx-server-certs/tls.crt;
ssl_certificate_key /etc/nginx-server-certs/tls.key;
}

}

EOF
```

3. Create a Kubernetes ConfigMap to hold the configuration of the NGINX server:

```
$ kubectl create configmap nginx-configmap --from-file=nginx.conf=./nginx.conf
```

4. Deploy the NGINX server:

```
$ cat <<EOF | istioctl kube-inject -f - | kubectl apply -f
-
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
  - port: 443
    protocol: TCP
  selector:
    run: my-nginx
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
```

```
selector:
  matchLabels:
    run: my-nginx
replicas: 1
template:
  metadata:
    labels:
      run: my-nginx
spec:
  containers:
    - name: my-nginx
      image: nginx
      ports:
        - containerPort: 443
  volumeMounts:
    - name: nginx-config
      mountPath: /etc/nginx
      readOnly: true
    - name: nginx-server-certs
      mountPath: /etc/nginx-server-certs
```

```
    readOnly: true
  volumes:
    - name: nginx-config
      configMap:
        name: nginx-configmap
    - name: nginx-server-certs
      secret:
        secretName: nginx-server-certs
```

EOF

5. To test that the NGINX server was deployed successfully, send a request to the server from its sidecar proxy without checking the server's certificate (use the `-k` option of `curl`). Ensure that the server's certificate is printed correctly, i.e., common name (CN) is equal to `nginx.example.com`.

```
$ kubectl exec "$(kubectl get pod -l run=my-nginx -o jsonpath={.items..metadata.name})" -c istio-proxy -- curl -sS -v -k --resolve nginx.example.com:443:127.0.0.1 https://nginx.example.com
...
SSL connection using TLSv1.2 / ECDHE-RSA-AES256-GCM-SHA384
ALPN, server accepted to use http/1.1
Server certificate:
  subject: CN=nginx.example.com; O=some organization
  start date: May 27 14:18:47 2020 GMT
  expire date: May 27 14:18:47 2021 GMT
  issuer: O=example Inc.; CN=example.com
  SSL certificate verify result: unable to get local issuer
  certificate (20), continuing anyway.

> GET / HTTP/1.1
> User-Agent: curl/7.58.0
> Host: nginx.example.com
...
< HTTP/1.1 200 OK
```

```
< Server: nginx/1.17.10
...
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

# Configure an ingress gateway

1. Define a Gateway with a server section for port 443.

Note the PASSTHROUGH TLS mode which instructs the gateway to pass the ingress traffic AS IS, without terminating TLS.

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: mygateway
spec:
  selector:
    istio: ingressgateway # use istio default ingress gateway
  servers:
  - port:
      number: 443
      name: https
      protocol: HTTPS
    tls:
      mode: PASSTHROUGH
  hosts:
  - nginx.example.com
EOF
```

## 2. Configure routes for traffic entering via the Gateway:

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: nginx
spec:
  hosts:
  - nginx.example.com
  gateways:
  - mygateway
  tls:
  - match:
    - port: 443
      sniHosts:
      - nginx.example.com
  route:
```

```
- destination:  
    host: my-nginx  
    port:  
        number: 443  
EOF
```

3. Follow the instructions in Determining the ingress IP and ports to define the `SECURE_INGRESS_PORT` and `INGRESS_HOST` environment variables.
4. Access the NGINX service from outside the cluster. Note that the correct certificate is returned by the server and it is successfully verified (*SSL certificate verify ok* is printed).

```
$ curl -v --resolve "nginx.example.com:$SECURE_INGRESS_PORT  
:$INGRESS_HOST" --cacert example.com.crt "https://nginx.ex  
ample.com:$SECURE_INGRESS_PORT"
```

Server certificate:

```
subject: CN=nginx.example.com; O=some organization  
start date: Wed, 15 Aug 2018 07:29:07 GMT  
expire date: Sun, 25 Aug 2019 07:29:07 GMT  
issuer: O=example Inc.; CN=example.com  
SSL certificate verify ok.
```

```
< HTTP/1.1 200 OK  
< Server: nginx/1.15.2  
...  
<html>  
<head>  
<title>Welcome to nginx!</title>
```

# Cleanup

## 1. Remove created Kubernetes resources:

```
$ kubectl delete secret nginx-server-certs  
$ kubectl delete configmap nginx-configmap  
$ kubectl delete service my-nginx  
$ kubectl delete deployment my-nginx  
$ kubectl delete gateway mygateway  
$ kubectl delete virtualservice nginx
```

## 2. Delete the certificates and keys:

```
$ rm example.com.crt example.com.key nginx.example.com.crt  
nginx.example.com.key nginx.example.com.csr
```

3. Delete the generated configuration files used in this example:

```
$ rm ./nginx.conf
```

# Kubernetes Ingress

⌚ 3 minute read ✓ page test

---

This task describes how to configure Istio to expose a service outside of the service mesh cluster, using the Kubernetes Ingress Resource.

Using the Istio Gateway, rather than Ingress, is recommended to make use of the full feature

set that Istio offers, such as rich traffic management and security features.

## Before you begin

Follow the instructions in the [Before you begin](#) and [Determining the ingress IP and ports](#) sections of the Ingress Gateways task.

# Configuring ingress using an Ingress resource

A Kubernetes Ingress Resources exposes HTTP and HTTPS routes from outside the cluster to services within the cluster.

Let's see how you can configure a Ingress on port 80 for HTTP traffic.

1. Create an Ingress resource:

```
$ kubectl apply -f - <<EOF
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: istio
  name: ingress
spec:
  rules:
  - host: httpbin.example.com
    http:
      paths:
      - path: /status/*
        backend:
          serviceName: httpbin
          servicePort: 8000
EOF
```

The `kubernetes.io/ingress.class` annotation is

required to tell the Istio gateway controller that it should handle this Ingress, otherwise it will be ignored.

## 2. Access the *httpbin* service using *curl*:

```
$ curl -s -I -HHost:httpbin.example.com "http://$INGRESS_HO
ST:$INGRESS_PORT/status/200"
HTTP/1.1 200 OK
server: istio-envoy
...
...
```

Note that you use the `-H` flag to set the *Host* HTTP header to “`httpbin.example.com`”. This is needed because the Ingress is configured to handle “`httpbin.example.com`”, but in your test

environment you have no DNS binding for that host and are simply sending your request to the ingress IP.

3. Access any other URL that has not been explicitly exposed. You should see an HTTP 404 error:

```
$ curl -s -I -HHost:httpbin.example.com "http://$INGRESS_HO  
ST:$INGRESS_PORT/headers"  
HTTP/1.1 404 Not Found  
...
```

## Next Steps

# TLS

Ingress supports specifying TLS settings. This is supported by Istio, but the referenced Secret must exist in the namespace of the `istio-ingressgateway` deployment (typically `istio-system`). cert-manager can be used to generate these certificates.

## Specifying path type

By default, Istio will treat paths as exact matches,

unless they end in `/*` or `.*`, in which case they will become prefix matches. Other regular expressions are not supported.

In Kubernetes 1.18, a new field, `pathType`, was added. This allows explicitly declaring a path as `Exact` or `Prefix`.

## Specifying IngressClass

In Kubernetes 1.18, a new resource, `IngressClass`, was added, replacing the `kubernetes.io/ingress.class`

annotation on the Ingress resource. If you are using this resource, you will need to set the controller field to `istio.io/ingress-controller`. For example:

```
apiVersion: networking.k8s.io/v1beta1
kind: IngressClass
metadata:
  name: istio
spec:
  controller: istio.io/ingress-controller
---
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress
spec:
  ingressClassName: istio
  rules:
```

```
- host: httpbin.example.com
  http:
    paths:
      - path: /
        pathType: Prefix
        backend:
          serviceName: httpbin
          servicePort: 8000
```

# Cleanup

Delete the Ingress configuration, and shutdown the httpbin service:

```
$ kubectl delete ingress ingress
$ kubectl delete --ignore-not-found=true -f @samples/httpbin/htt
pbin.yaml@
```

# Kubernetes Gateway API

⌚ 2 minute read ✓ page test

---

This task describes how to configure Istio to expose a service outside of the service mesh cluster, using the Kubernetes Gateway API. These APIs are an actively developed evolution of the Kubernetes Service and Ingress APIs.

# Setup

1. Install the Gateway API CRDs:

```
$ kubectl kustomize "github.com/kubernetes-sigs/gateway-api  
/config/crd?ref=v0.3.0" | kubectl apply -f -
```

2. Install Istio:

```
$ istioctl install
```

3. Follow the instructions in the Determining the ingress IP and ports sections of the Ingress Gateways task in order to retrieve the external IP address of your

ingress gateway.

# Configuring a Gateway

See the Gateway API documentation for information about the APIs.

1. Deploy a test application:

```
$ kubectl apply -f @samples/httpbin/httpbin.yaml@
```

2. Deploy the Gateway API configuration:

```
$ kubectl apply -f - <<EOF
apiVersion: networking.x-k8s.io/v1alpha1
kind: GatewayClass
metadata:
  name: istio
spec:
  controller: istio.io/gateway-controller
---
apiVersion: networking.x-k8s.io/v1alpha1
kind: Gateway
metadata:
  name: gateway
  namespace: istio-system
spec:
  gatewayClassName: istio
  listeners:
  - hostname: "*"
    port: 80
    protocol: HTTP
  routes:
```

```
namespaces:  
  from: All  
selector:  
  matchLabels:  
    selected: "yes"  
kind: HTTPRoute  
---  
apiVersion: networking.x-k8s.io/v1alpha1  
kind: HTTPRoute  
metadata:  
  name: http  
  namespace: default  
  labels:  
    selected: "yes"  
spec:  
  gateways:  
    allow: All  
  hostnames: ["httpbin.example.com"]  
  rules:  
  - matches:
```

```
- path:  
    type: Prefix  
    value: /get  
  
filters:  
- type: RequestHeaderModifier  
  requestHeaderModifier:  
    add:  
      my-added-header: added-value  
  
forwardTo:  
- serviceName: httpbin  
  port: 8000  
  
EOF
```

### 3. Access the *httpbin* service using *curl*:

```
$ curl -s -I -HHost:httpbin.example.com "http://$INGRESS_HOST:$INGRESS_PORT/get"
HTTP/1.1 200 OK
server: istio-envoy
...
...
```

Note the use of the `-H` flag to set the *Host* HTTP header to “`httpbin.example.com`”. This is needed because the `HTTPRoute` is configured to handle “`httpbin.example.com`”, but in your test environment you have no DNS binding for that host and are simply sending your request to the ingress IP.

4. Access any other URL that has not been explicitly exposed. You should see an HTTP 404 error:

```
$ curl -s -I -HHost:httpbin.example.com "http://$INGRESS_HOST:$INGRESS_PORT/headers"
HTTP/1.1 404 Not Found
...

```

# Accessing External Services

⌚ 11 minute read ✓ page test

---

Because all outbound traffic from an Istio-enabled pod is redirected to its sidecar proxy by default, accessibility of URLs outside of the cluster depends on the configuration of the proxy. By default, Istio configures the Envoy proxy to pass through requests

for unknown services. Although this provides a convenient way to get started with Istio, configuring stricter control is usually preferable.

This task shows you how to access external services in three different ways:

1. Allow the Envoy proxy to pass requests through to services that are not configured inside the mesh.
2. Configure service entries to provide controlled access to external services.
3. Completely bypass the Envoy proxy for a specific

range of IPs.

# Before you begin

- Set up Istio by following the instructions in the Installation guide. Use the demo configuration profile or otherwise enable Envoy's access logging.
- Deploy the sleep sample app to use as a test source for sending requests. If you have automatic sidecar injection enabled, run the following command to deploy the sample app:

```
$ kubectl apply -f @samples/sleep/sleep.yaml@
```

Otherwise, manually inject the sidecar before deploying the sleep application with the following command:

```
$ kubectl apply -f <(istioctl kube-inject -f @samples/sleep/sleep.yaml@)
```



You can use any pod with curl installed as a test source.

- Set the SOURCE\_POD environment variable to the

name of your source pod:

```
$ export SOURCE_POD=$(kubectl get pod -l app=sleep -o jsonpath='{.items..metadata.name}')
```

## Envoy passthrough to external services

Istio has an installation option,  
meshConfig.outboundTrafficPolicy.mode, that configures  
the sidecar handling of external services, that is,

those services that are not defined in Istio's internal service registry. If this option is set to `ALLOW_ANY`, the Istio proxy lets calls to unknown services pass through. If the option is set to `REGISTRY_ONLY`, then the Istio proxy blocks any host without an HTTP service or service entry defined within the mesh. `ALLOW_ANY` is the default value, allowing you to start evaluating Istio quickly, without controlling access to external services. You can then decide to configure access to external services **later**.

1. To see this approach in action you need to ensure that your Istio installation is configured with the `meshConfig.outboundTrafficPolicy.mode` option set to

`ALLOW_ANY`. Unless you explicitly set it to `REGISTRY_ONLY` mode when you installed Istio, it is probably enabled by default.

Run the following command to verify that `meshConfig.outboundTrafficPolicy.mode` option is set to `ALLOW_ANY` or is omitted:

```
$ kubectl get istiooperator installed-state -n istio-system  
-o jsonpath='{.spec.meshConfig.outboundTrafficPolicy.mode}'  
  
ALLOW_ANY
```

You should either see `ALLOW_ANY` or no output (default `ALLOW_ANY`).



If you have explicitly configured REGISTRY\_ONLY mode, you can change it by rerunning your original `istioctl install` command with the changed setting, for example:

```
$ istioctl install <flags-you-used-to-install-Istio> --set meshConfig.outboundTrafficPolicy.mode=ALLOW_ANY
```

2. Make a couple of requests to external HTTPS services from the SOURCE\_POD to confirm successful 200 responses:

```
$ kubectl exec "$SOURCE_POD" -c sleep -- curl -sSI https://www.google.com | grep "HTTP/"; kubectl exec "$SOURCE_POD" -c sleep -- curl -sI https://edition.cnn.com | grep "HTTP/"  
HTTP/2 200  
HTTP/2 200
```

Congratulations! You successfully sent egress traffic from your mesh.

This simple approach to access external services, has the drawback that you lose Istio monitoring and control for traffic to external services. The next section shows you how to monitor and control your mesh's access to external services.

# **Controlled access to external services**

Using Istio `ServiceEntry` configurations, you can access any publicly accessible service from within your Istio cluster. This section shows you how to configure access to an external HTTP service, `httpbin.org`, as well as an external HTTPS service, `www.google.com` without losing Istio's traffic monitoring and control features.

## **Change to the `blocking-by-`**

# default policy

To demonstrate the controlled way of enabling access to external services, you need to change the `meshConfig.outboundTrafficPolicy.mode` option from the `ALLOW_ANY` mode to the `REGISTRY_ONLY` mode.

You can add controlled access to services that are already accessible in `ALLOW_ANY` mode. This way, you can start using Istio features on some external services without blocking any others. Once you've configured all of



your services, you can then switch the mode to `REGISTRY_ONLY` to block any other unintentional accesses.

1. Change the `meshConfig.outboundTrafficPolicy.mode` option to `REGISTRY_ONLY`.

If you used an `IstioOperator` CR to install Istio, add the following field to your configuration:

```
spec:  
  meshConfig:  
    outboundTrafficPolicy:  
      mode: REGISTRY_ONLY
```

Otherwise, add the equivalent setting to your original `istioctl` install command, for example:

```
$ istioctl install <flags-you-used-to-install-Istio> \
    --set meshConfig.outboundTrafficPolicy.m
ode=REGISTRY_ONLY
```

2. Make a couple of requests to external HTTPS services from `SOURCE_POD` to verify that they are now blocked:

```
$ kubectl exec "$SOURCE_POD" -c sleep -- curl -sI https://w
ww.google.com | grep "HTTP/"; kubectl exec "$SOURCE_POD" -
c sleep -- curl -sI https://edition.cnn.com | grep "HTTP/"
command terminated with exit code 35
command terminated with exit code 35
```



It may take a while for the configuration change to propagate, so you might still get successful connections. Wait for several seconds and then retry the last command.

## Access an external HTTP service

1. Create a `ServiceEntry` to allow access to an external HTTP service.

DNS resolution is used in the service entry below as a security measure. Setting the resolution to `NONE` opens a possibility for attack. A malicious client could pretend that it's accessing `httpbin.org` by setting it in the `HOST` header, while really connecting to a different IP (that is not associated with `httpbin.org`). The Istio sidecar proxy will trust the `HOST` header, and incorrectly allow the traffic,



even though it is being delivered to the IP address of a different host. That host can be a malicious site, or a legitimate site, prohibited by the mesh security policies.

With DNS resolution, the sidecar proxy will ignore the original destination IP address and direct the traffic to [httpbin.org](http://httpbin.org), performing a DNS query to get an IP address of [httpbin.org](http://httpbin.org).

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: httpbin-ext
spec:
  hosts:
  - httpbin.org
  ports:
  - number: 80
    name: http
    protocol: HTTP
  resolution: DNS
  location: MESH_EXTERNAL
EOF
```

2. Make a request to the external HTTP service from SOURCE\_POD:

```
$ kubectl exec "$SOURCE_POD" -c sleep -- curl -sS http://httpbin.org/headers
{
  "headers": {
    "Accept": "*/*",
    "Host": "httpbin.org",
    ...
    "X-Envoy-Decorator-Operation": "httpbin.org:80/*",
    ...
  }
}
```

Note the headers added by the Istio sidecar proxy: X-Envoy-Decorator-Operation.

3. Check the log of the sidecar proxy of SOURCE\_POD:

```
$ kubectl logs "$SOURCE_POD" -c istio-proxy | tail  
[2019-01-24T12:17:11.640Z] "GET /headers HTTP/1.1" 200 - 0  
599 214 214 "-" "curl/7.60.0" "17fde8f7-fa62-9b39-8999-3023  
24e6def2" "httpbin.org" "35.173.6.94:80" outbound|80||httpb  
in.org - 35.173.6.94:80 172.30.109.82:55314 -
```

Note the entry related to your HTTP request to httpbin.org/headers.

## Access an external HTTPS service

1. Create a ServiceEntry to allow access to an

## external HTTPS service.

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: google
spec:
  hosts:
  - www.google.com
  ports:
  - number: 443
    name: https
    protocol: HTTPS
  resolution: DNS
  location: MESH_EXTERNAL
EOF
```

2. Make a request to the external HTTPS service

from SOURCE\_POD:

```
$ kubectl exec "$SOURCE_POD" -c sleep -- curl -sSI https://www.google.com | grep "HTTP/"  
HTTP/2 200
```

### 3. Check the log of the sidecar proxy of SOURCE\_POD:

```
$ kubectl logs "$SOURCE_POD" -c istio-proxy | tail  
[2019-01-24T12:48:54.977Z] "-" 0 - 601 17766 1289 - "-"  
"-" "-" "-" "172.217.161.36:443" outbound|443||www.google.  
com 172.30.109.82:59480 172.217.161.36:443 172.30.109.82:59  
478 www.google.com
```

Note the entry related to your HTTPS request to www.google.com.

# Manage traffic to external services

Similar to inter-cluster requests, Istio routing rules can also be set for external services that are accessed using `ServiceEntry` configurations. In this example, you set a timeout rule on calls to the `httpbin.org` service.

1. From inside the pod being used as the test source, make a *curl* request to the `/delay` endpoint of the `httpbin.org` external service:

```
$ kubectl exec "$SOURCE_POD" -c sleep -- time curl -o /dev/null -sS -w "%{http_code}\n" http://httpbin.org/delay/5  
200  
real    0m5.024s  
user    0m0.003s  
sys     0m0.003s
```

The request should return 200 (OK) in approximately 5 seconds.

2. Use `kubectl` to set a 3s timeout on calls to the `httpbin.org` external service:

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: httpbin-ext
spec:
  hosts:
    - httpbin.org
  http:
    - timeout: 3s
      route:
        - destination:
            host: httpbin.org
            weight: 100
EOF
```

3. Wait a few seconds, then make the *curl* request again:

```
$ kubectl exec "$SOURCE_POD" -c sleep -- time curl -o /dev/null -sS -w "%{http_code}\n" http://httpbin.org/delay/5  
504  
real    0m3.149s  
user    0m0.004s  
sys     0m0.004s
```

This time a 504 (Gateway Timeout) appears after 3 seconds. Although httpbin.org was waiting 5 seconds, Istio cut off the request at 3 seconds.

## Cleanup the controlled access to external services

```
$ kubectl delete serviceentry httpbin-ext google
$ kubectl delete virtualservice httpbin-ext --ignore-not-found=true
```

## **Direct access to external services**

If you want to completely bypass Istio for a specific IP range, you can configure the Envoy sidecars to prevent them from intercepting external requests. To set up the bypass, change either the

`global.proxy.includeIPRanges` or the  
`global.proxy.excludeIPRanges` configuration option and  
update the `istio-sidecar-injector` configuration map  
using the `kubectl apply` command. This can also be  
configured on a pod by setting corresponding  
annotations such as

`traffic.sidecar.istio.io/includeOutboundIPRanges`. After  
updating the `istio-sidecar-injector` configuration, it  
affects all future application pod deployments.

Unlike Envoy passthrough to external services,  
which uses the `ALLOW_ANY` traffic policy to

 instruct the Istio sidecar proxy to passthrough calls to unknown services, this approach completely bypasses the sidecar, essentially disabling all of Istio's features for the specified IPs. You cannot incrementally add service entries for specific destinations, as you can with the `ALLOW_ANY` approach. Therefore, this configuration approach is only recommended as a last resort when, for performance or other reasons, external access cannot be configured using the sidecar.

A simple way to exclude all external IPs from being redirected to the sidecar proxy is to set the `global.proxy.includeIPRanges` configuration option to the IP range or ranges used for internal cluster services. These IP range values depend on the platform where your cluster runs.

## Determine the internal IP ranges for your platform

Set the value of `values.global.proxy.includeIPRanges` according to your cluster provider.

# IBM Cloud Private

1. Get your `service_cluster_ip_range` from IBM Cloud Private configuration file under `cluster/config.yaml`:

```
$ grep service_cluster_ip_range cluster/config.yaml
```

The following is a sample output:

```
service_cluster_ip_range: 10.0.0.1/24
```

2. Use `--set`  
`values.global.proxy.includeIPRanges="10.0.0.1/24"`

# IBM Cloud Kubernetes Service

Use --set

```
values.global.proxy.includeIPRanges="172.30.0.0/16\,172.  
21.0.0/16\,10.10.10.0/24"
```

## Google Container Engine (GKE)

The ranges are not fixed, so you will need to run the gcloud container clusters describe command to determine the ranges to use. For example:

```
$ gcloud container clusters describe XXXXXXXX --zone=XXXXXX | grep -e clusterIpv4Cidr -e servicesIpv4Cidr
clusterIpv4Cidr: 10.4.0.0/14
servicesIpv4Cidr: 10.7.240.0/20
```

Use --set

```
values.global.proxy.includeIPRanges="10.4.0.0/14\,10.7.240.0/20"
```

## Azure Container Service(ACS)

Use --set

```
values.global.proxy.includeIPRanges="10.244.0.0/16\,10.240.0.0/16"
```

# Minikube, Docker For Desktop, Bare Metal

The default value is `10.96.0.0/12`, but it's not fixed.  
Use the following command to determine your actual value:

```
$ kubectl describe pod kube-apiserver -n kube-system | grep 'service-cluster-ip-range'  
--service-cluster-ip-range=10.96.0.0/12
```

Use `--set`

```
values.global.proxy.includeIPRanges="10.96.0.0/12"
```

# Configuring the proxy bypass

- ⚠ Remove the service entry and virtual service previously deployed in this guide.
- Update your `istio-sidecar-injector` configuration map using the IP ranges specific to your platform. For example, if the range is `10.0.0.1/24`, use the following command:

```
$ istioctl install <flags-you-used-to-install-Istio> --set value  
s.global.proxy.includeIPRanges="10.0.0.1/24"
```

Use the same command that you used to install Istio and add --set values.global.proxy.includeIPRanges="10.0.0.1/24".

## Access the external services

Because the bypass configuration only affects new deployments, you need to terminate and then redeploy the sleep application as described in the

Before you begin **section**.

After updating the `istio-sidecar-injector` configmap and redeploying the `sleep` application, the Istio sidecar will only intercept and manage internal requests within the cluster. Any external request bypasses the sidecar and goes straight to its intended destination. For example:

```
$ kubectl exec "$SOURCE_POD" -c sleep -- curl -sS http://httpbin.org/headers
{
  "headers": {
    "Accept": "*/*",
    "Host": "httpbin.org",
    ...
  }
}
```

Unlike accessing external services through HTTP or HTTPS, you don't see any headers related to the Istio sidecar and the requests sent to external services do not appear in the log of the sidecar. Bypassing the Istio sidecars means you can no longer monitor the access to external services.

# Cleanup the direct access to external services

Update the configuration to stop bypassing sidecar proxies for a range of IPs:

```
$ istioctl install <flags-you-used-to-install-Istio>
```

# Understanding what happened

In this task you looked at three ways to call external services from an Istio mesh:

1. Configuring Envoy to allow access to any external service.
2. Use a service entry to register an accessible external service inside the mesh. This is the recommended approach.
3. Configuring the Istio sidecar to exclude external IPs from its remapped IP table.

The first approach directs traffic through the Istio sidecar proxy, including calls to services that are

unknown inside the mesh. When using this approach, you can't monitor access to external services or take advantage of Istio's traffic control features for them. To easily switch to the second approach for specific services, simply create service entries for those external services. This process allows you to initially access any external service and then later decide whether or not to control access, enable traffic monitoring, and use traffic control features as needed.

The second approach lets you use all of the same Istio service mesh features for calls to services inside or outside of the cluster. In this task, you learned how to

monitor access to external services and set a timeout rule for calls to an external service.

The third approach bypasses the Istio sidecar proxy, giving your services direct access to any external server. However, configuring the proxy this way does require cluster-provider specific knowledge and configuration. Similar to the first approach, you also lose monitoring of access to external services and you can't apply Istio features on traffic to external services.

# Security note

Note that configuration examples in this task  
**do not enable secure egress traffic**



**control** in Istio. A malicious application can bypass the Istio sidecar proxy and access any external service without Istio control.

To implement egress traffic control in a more secure way, you must direct egress traffic through an egress

gateway and review the security concerns described in the additional security considerations section.

## Cleanup

Shutdown the sleep service:

```
$ kubectl delete -f @samples/sleep/sleep.yaml@
```



# Egress TLS Origination

⌚ 6 minute read ✓ page test

---

The Accessing External Services task demonstrates how external, i.e., outside of the service mesh, HTTP and HTTPS services can be accessed from applications inside the mesh. As described in that task, a ServiceEntry is used to configure Istio to access

external services in a controlled way. This example shows how to configure Istio to perform TLS origination for traffic to an external service. Istio will open HTTPS connections to the external service while the original traffic is HTTP.

## Use case

Consider a legacy application that performs HTTP calls to external sites. Suppose the organization that operates the application receives a new requirement

which states that all the external traffic must be encrypted. With Istio, this requirement can be achieved just by configuration, without changing any code in the application. The application can send unencrypted HTTP requests and Istio will then encrypt them for the application.

Another benefit of sending unencrypted HTTP requests from the source, and letting Istio perform the TLS upgrade, is that Istio can produce better telemetry and provide more routing control for requests that are not encrypted.

# Before you begin

- Setup Istio by following the instructions in the Installation guide.
- Start the sleep sample which will be used as a test source for external calls.

If you have enabled automatic sidecar injection, deploy the sleep application:

```
$ kubectl apply -f @samples/sleep/sleep.yaml@
```

Otherwise, you have to manually inject the sidecar before deploying the sleep application:

```
$ kubectl apply -f <(istioctl kube-inject -f @samples/sleep/sleep.yaml@)
```

Note that any pod that you can `exec` and `curl` from will do for the procedures below.

- Create a shell variable to hold the name of the source pod for sending requests to external services. If you used the `sleep` sample, run:

```
$ export SOURCE_POD=$(kubectl get pod -l app=sleep -o jsonpath='{.items..metadata.name}')
```

# Configuring access to an external service

First start by configuring access to an external service, `edition.cnn.com`, using the same technique shown in the Accessing External Services task. This time, however, use a single `ServiceEntry` to enable both HTTP and HTTPS access to the service.

1. Create a `ServiceEntry` to enable access to `edition.cnn.com`:

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: edition-cnn-com
spec:
  hosts:
  - edition.cnn.com
  ports:
  - number: 80
    name: http-port
    protocol: HTTP
  - number: 443
    name: https-port
    protocol: HTTPS
  resolution: DNS
EOF
```

2. Make a request to the external HTTP service:

```
$ kubectl exec "${SOURCE_POD}" -c sleep -- curl -sSL -o /de  
v/null -D - http://edition.cnn.com/politics  
HTTP/1.1 301 Moved Permanently  
...  
location: https://edition.cnn.com/politics  
...  
  
HTTP/2 200  
...
```

The output should be similar to the above (some details replaced by ellipsis).

Notice the `-L` flag of *curl* which instructs *curl* to follow redirects. In this case, the server returned a redirect response (301 Moved Permanently) for the HTTP request to `http://edition.cnn.com/politics`. The redirect

response instructs the client to send an additional request, this time using HTTPS, to <https://edition.cnn.com/politics>. For the second request, the server returned the requested content and a *200 OK* status code.

Although the *curl* command handled the redirection transparently, there are two issues here. The first issue is the redundant request, which doubles the latency of fetching the content of

<http://edition.cnn.com/politics>. The second issue is that the path of the URL, *politics* in this case, is sent in clear text. If there is an attacker who sniffs the communication between your application and

`edition.cnn.com`, the attacker would know which specific topics of `edition.cnn.com` the application fetched. For privacy reasons, you might want to prevent such disclosure.

Both of these issues can be resolved by configuring Istio to perform TLS origination.

## **TLS origination for egress traffic**

1. Redefine your ServiceEntry from the previous section to redirect HTTP requests to port 443 and add a DestinationRule to perform TLS origination:

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: edition-cnn-com
spec:
  hosts:
  - edition.cnn.com
  ports:
  - number: 80
    name: http-port
    protocol: HTTP
    targetPort: 443
  - number: 443
    name: https-port
```

```
    protocol: HTTPS
resolution: DNS
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: edition-cnn-com
spec:
  host: edition.cnn.com
  trafficPolicy:
    portLevelSettings:
      - port:
          number: 80
        tls:
          mode: SIMPLE # initiates HTTPS when accessing editi
on.cnn.com
EOF
```

The above DestinationRule will perform TLS

origination for HTTP requests on port 80 and the ServiceEntry will then redirect the requests on port 80 to target port 443.

## 2. Send an HTTP request to

`http://edition.cnn.com/politics`, as in the previous section:

```
$ kubectl exec "${SOURCE_POD}" -c sleep -- curl -ssl -o /dev/null -D - http://edition.cnn.com/politics
HTTP/1.1 200 OK
...
```

This time you receive *200 OK* as the first and the only response. Istio performed TLS origination for *curl* so the original HTTP request was forwarded

to `edition.cnn.com` as HTTPS. The server returned the content directly, without the need for redirection. You eliminated the double round trip between the client and the server, and the request left the mesh encrypted, without disclosing the fact that your application fetched the *politics* section of `edition.cnn.com`.

Note that you used the same command as in the previous section. For applications that access external services programmatically, the code does not need to be changed. You get the benefits of TLS origination by configuring Istio, without changing a line of code.

3. Note that the applications that used HTTPS to access the external service continue to work as before:

```
$ kubectl exec "${SOURCE_POD}" -c sleep -- curl -sSL -o /de  
v/null -D - https://edition.cnn.com/politics  
HTTP/2 200  
...
```

## Additional security considerations

Because the traffic between the application pod and the sidecar proxy on the local host is still unencrypted, an attacker that is able to penetrate the node of your application would still be able to see the unencrypted communication on the local network of the node. In some environments a strict security requirement might state that all the traffic must be encrypted, even on the local network of the nodes. With such a strict requirement, applications should use HTTPS (TLS) only. The TLS origination described in this example would not be sufficient.

Also note that even with HTTPS originated by the application, an attacker could know that requests to

`edition.cnn.com` are being sent by inspecting Server Name Indication (SNI). The *SNI* field is sent unencrypted during the TLS handshake. Using HTTPS prevents the attackers from knowing specific topics and articles but does not prevent an attacker from learning that `edition.cnn.com` is accessed.

## Cleanup

1. Remove the Istio configuration items you created:

```
$ kubectl delete serviceentry edition-cnn-com  
$ kubectl delete destinationrule edition-cnn-com
```

## 2. Shutdown the sleep service:

```
$ kubectl delete -f @samples/sleep/sleep.yaml@
```



# Egress Gateways

⌚ 13 minute read ✓ page test

---



This example does not work in Minikube.

The Accessing External Services task shows how to configure Istio to allow access to external HTTP and HTTPS services from applications inside the mesh.

There, the external services are called directly from the client sidecar. This example also shows how to configure Istio to call external services, although this time indirectly via a dedicated *egress gateway* service.

Istio uses ingress and egress gateways to configure load balancers executing at the edge of a service mesh. An ingress gateway allows you to define entry points into the mesh that all incoming traffic flows through. Egress gateway is a symmetrical concept; it defines exit points from the mesh. Egress gateways allow you to apply Istio features, for example, monitoring and route rules, to traffic exiting the mesh.

# Use case

Consider an organization that has a strict security requirement that all traffic leaving the service mesh must flow through a set of dedicated nodes. These nodes will run on dedicated machines, separated from the rest of the nodes running applications in the cluster. These special nodes will serve for policy enforcement on the egress traffic and will be monitored more thoroughly than other nodes.

Another use case is a cluster where the application nodes don't have public IPs, so the in-mesh services

that run on them cannot access the Internet. Defining an egress gateway, directing all the egress traffic through it, and allocating public IPs to the egress gateway nodes allows the application nodes to access external services in a controlled way.

## Before you begin

- Setup Istio by following the instructions in the Installation guide.



The egress gateway and access logging will be enabled if you install the demo configuration profile.

- Deploy the sleep sample app to use as a test source for sending requests. If you have automatic sidecar injection enabled, run the following command to deploy the sample app:

```
$ kubectl apply -f @samples/sleep/sleep.yaml@
```

Otherwise, manually inject the sidecar before deploying the sleep application with the following

command:

```
$ kubectl apply -f <(istioctl kube-inject -f @samples/sleep/sleep.yaml@)
```



You can use any pod with `curl` installed as a test source.

- Set the `SOURCE_POD` environment variable to the name of your source pod:

```
$ export SOURCE_POD=$(kubectl get pod -l app=sleep -o jsonpath={.items..metadata.name})
```

- Enable Envoy's access logging

The instructions in this task create a destination rule for the egress gateway in the `default` namespace and assume that the client, `SOURCE_POD`, is also running in the `default` namespace. If not, the destination rule will not be found on the destination rule lookup path and the client requests will fail.

# Deploy Istio egress gateway

1. Check if the Istio egress gateway is deployed:

```
$ kubectl get pod -l istio=egressgateway -n istio-system
```

If no pods are returned, deploy the Istio egress gateway by performing the following step.

2. If you used an `IstioOperator` CR to install Istio, add the following fields to your configuration:

```
spec:  
  components:  
    egressGateways:  
      - name: istio-egressgateway  
        enabled: true
```

Otherwise, add the equivalent settings to your original `istioctl install` command, for example:

```
$ istioctl install <flags-you-used-to-install-Istio> \  
          --set components.egressGateways[0].name=  
istio-egressgateway \  
          --set components.egressGateways[0].enabl  
ed=true
```

# Egress gateway for HTTP traffic

First create a `ServiceEntry` to allow direct traffic to an external service.

1. Define a `ServiceEntry` for `edition.cnn.com`.

DNS resolution must be used in the service entry below. If the resolution is `NONE`, the gateway will direct the traffic to itself in an infinite loop. This is because



the gateway receives a request with the original destination IP address which is equal to the service IP of the gateway (since the request is directed by sidecar proxies to the gateway).

With `DNS` resolution, the gateway performs a DNS query to get an IP address of the external service and directs the traffic to that IP address.

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: cnn
spec:
  hosts:
  - edition.cnn.com
  ports:
  - number: 80
    name: http-port
    protocol: HTTP
  - number: 443
    name: https
    protocol: HTTPS
  resolution: DNS
EOF
```

2. Verify that your ServiceEntry was applied correctly

by sending an HTTP request to  
`http://edition.cnn.com/politics.`

```
$ kubectl exec "$SOURCE_POD" -c sleep -- curl -sSL -o /dev/null -D - http://edition.cnn.com/politics
...
HTTP/1.1 301 Moved Permanently
...
location: https://edition.cnn.com/politics
...

HTTP/2 200
Content-Type: text/html; charset=utf-8
...
```

The output should be the same as in the TLS  
Origination for Egress Traffic **example**, without TLS

origination.

3. Create an egress Gateway for `edition.cnn.com`, port 80, and a destination rule for traffic directed to the egress gateway.

To direct multiple hosts through an egress gateway, you can include a list of hosts, or use `*` to match all, in the Gateway. The `subset` field in the DestinationRule should be reused for the additional hosts.



```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: istio-egressgateway
spec:
  selector:
    istio: egressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - edition.cnn.com
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: egressgateway-for-cnn
```

```
spec:  
  host: istio-egressgateway.istio-system.svc.cluster.local  
  subsets:  
    - name: cnn  
EOF
```

4. Define a `VirtualService` to direct traffic from the sidecars to the egress gateway and from the egress gateway to the external service:

```
$ kubectl apply -f - <<EOF  
apiVersion: networking.istio.io/v1alpha3  
kind: VirtualService  
metadata:  
  name: direct-cnn-through-egress-gateway  
spec:  
  hosts:  
    - edition.cnn.com
```

```
gateways:
- istio-egressgateway
- mesh
http:
- match:
  - gateways:
    - mesh
    port: 80
route:
- destination:
  host: istio-egressgateway.istio-system.svc.cluster.
local
  subset: cnn
  port:
    number: 80
  weight: 100
- match:
  - gateways:
    - istio-egressgateway
    port: 80
```

```
route:  
  - destination:  
    host: edition.cnn.com  
    port:  
      number: 80  
    weight: 100  
EOF
```

5. Resend the HTTP request to  
[http://edition.cnn.com/politics.](http://edition.cnn.com/politics)

```
$ kubectl exec "$SOURCE_POD" -c sleep -- curl -sSL -o /dev/null -D - http://edition.cnn.com/politics
...
HTTP/1.1 301 Moved Permanently
...
location: https://edition.cnn.com/politics
...
HTTP/2 200
Content-Type: text/html; charset=utf-8
...
```

The output should be the same as in the step 2.

6. Check the log of the `istio-egressgateway` pod for a line corresponding to our request. If Istio is deployed in the `istio-system` namespace, the command to print the log is:

```
$ kubectl logs -l istio=egressgateway -c istio-proxy -n istio-system | tail
```

You should see a line similar to the following:

```
[2019-09-03T20:57:49.103Z] "GET /politics HTTP/2" 301 - "-"  
"-" 0 0 90 89 "10.244.2.10" "curl/7.64.0" "ea379962-9b5c-4  
431-ab66-f01994f5a5a5" "edition.cnn.com" "151.101.65.67:80"  
outbound|80||edition.cnn.com - 10.244.1.5:80 10.244.2.10:5  
0482 edition.cnn.com -
```

Note that you only redirected the traffic from port 80 to the egress gateway. The HTTPS traffic to port 443 went directly to *edition.cnn.com*.

# Cleanup HTTP gateway

Remove the previous definitions before proceeding to the next step:

```
$ kubectl delete gateway istio-egressgateway  
$ kubectl delete serviceentry cnn  
$ kubectl delete virtualservice direct-cnn-through-egress-gatewa  
y  
$ kubectl delete destinationrule egressgateway-for-cnn
```

# Egress gateway for HTTPS

# traffic

In this section you direct HTTPS traffic (TLS originated by the application) through an egress gateway. You need to specify port 443 with protocol TLS in a corresponding ServiceEntry, an egress Gateway and a VirtualService.

1. Define a ServiceEntry for edition.cnn.com:

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: cnn
spec:
  hosts:
  - edition.cnn.com
  ports:
  - number: 443
    name: tls
    protocol: TLS
  resolution: DNS
EOF
```

2. Verify that your ServiceEntry was applied correctly by sending an HTTPS request to <https://edition.cnn.com/politics>.

```
$ kubectl exec "$SOURCE_POD" -c sleep -- curl -sSL -o /dev/null -D - https://edition.cnn.com/politics  
...  
HTTP/2 200  
Content-Type: text/html; charset=utf-8  
...
```

3. Create an egress Gateway for *edition.cnn.com*, a destination rule and a virtual service to direct the traffic through the egress gateway and from the egress gateway to the external service.

To direct multiple hosts through an egress gateway, you can include a list of hosts, or use \* to match all, in the



Gateway. The subset field in the DestinationRule should be reused for the additional hosts.

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: istio-egressgateway
spec:
  selector:
    istio: egressgateway
  servers:
  - port:
      number: 443
      name: tls
      protocol: TLS
```

```
hosts:
  - edition.cnn.com
tls:
  mode: PASSTHROUGH
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: egressgateway-for-cnn
spec:
  host: istio-egressgateway.istio-system.svc.cluster.local
  subsets:
    - name: cnn
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: direct-cnn-through-egress-gateway
spec:
  hosts:
```

```
- edition.cnn.com

gateways:
- mesh
- istio-egressgateway

tls:
- match:
  - gateways:
    - mesh
    port: 443
    sniHosts:
      - edition.cnn.com

route:
- destination:
  host: istio-egressgateway.istio-system.svc.cluster.local
    subset: cnn
    port:
      number: 443
- match:
  - gateways:
```

```
- istio-egressgateway
port: 443
sniHosts:
- edition.cnn.com
route:
- destination:
  host: edition.cnn.com
  port:
    number: 443
  weight: 100
EOF
```

#### 4. Send an HTTPS request to

<https://edition.cnn.com/politics>. The output should be the same as before.

```
$ kubectl exec "$SOURCE_POD" -c sleep -- curl -sSL -o /dev/null -D - https://edition.cnn.com/politics  
...  
HTTP/2 200  
Content-Type: text/html; charset=utf-8  
...
```

5. Check the log of the egress gateway's proxy. If Istio is deployed in the `istio-system` namespace, the command to print the log is:

```
$ kubectl logs -l istio=egressgateway -n istio-system
```

You should see a line similar to the following:

```
[2019-01-02T11:46:46.981Z] " - - - " 0 - 627 1879689 44 - "-"  
"- - -" "151.101.129.67:443" outbound|443||edition.cnn  
.com 172.30.109.80:41122 172.30.109.80:443 172.30.109.112:5  
9970 edition.cnn.com
```

# Cleanup HTTPS gateway

```
$ kubectl delete serviceentry cnn  
$ kubectl delete gateway istio-egressgateway  
$ kubectl delete virtualservice direct-cnn-through-egress-gatewa  
y  
$ kubectl delete destinationrule egressgateway-for-cnn
```

# Additional security considerations

Note that defining an egress `Gateway` in Istio does not in itself provide any special treatment for the nodes on which the egress gateway service runs. It is up to the cluster administrator or the cloud provider to deploy the egress gateways on dedicated nodes and to introduce additional security measures to make these nodes more secure than the rest of the mesh.

Istio *cannot securely enforce* that all egress traffic actually flows through the egress gateways. Istio only

enables such flow through its sidecar proxies. If attackers bypass the sidecar proxy, they could directly access external services without traversing the egress gateway. Thus, the attackers escape Istio's control and monitoring. The cluster administrator or the cloud provider must ensure that no traffic leaves the mesh bypassing the egress gateway. Mechanisms external to Istio must enforce this requirement. For example, the cluster administrator can configure a firewall to deny all traffic not coming from the egress gateway. The Kubernetes network policies can also forbid all the egress traffic not originating from the egress gateway (see the next section for an example).

Additionally, the cluster administrator or the cloud provider can configure the network to ensure application nodes can only access the Internet via a gateway. To do this, the cluster administrator or the cloud provider can prevent the allocation of public IPs to pods other than gateways and can configure NAT devices to drop packets not originating at the egress gateways.

## **Apply Kubernetes network policies**

This section shows you how to create a Kubernetes network policy to prevent bypassing of the egress gateway. To test the network policy, you create a namespace, `test-egress`, deploy the `sleep` sample to it, and then attempt to send requests to a gateway-secured external service.

1. Follow the steps in the Egress gateway for HTTPS traffic section.
2. Create the `test-egress` namespace:

```
$ kubectl create namespace test-egress
```

3. Deploy the `sleep` sample to the `test-egress`

namespace.

```
$ kubectl apply -n test-egress -f @samples/sleep/sleep.yaml  
@
```

4. Check that the deployed pod has a single container with no Istio sidecar attached:

```
$ kubectl get pod "$(kubectl get pod -n test-egress -l app=sleep -o jsonpath={.items..metadata.name})" -n test-egress  
NAME                  READY     STATUS    RESTARTS   AGE  
sleep-776b7bcdcd-z7mc4   1/1      Running   0          18m
```

5. Send an HTTPS request to

<https://edition.cnn.com/politics> from the sleep pod in the test-egress namespace. The request will

succeed since you did not define any restrictive policies yet.

```
$ kubectl exec "$(kubectl get pod -n test-egress -l app=sleep -o jsonpath={.items..metadata.name})" -n test-egress -c sleep -- curl -s -o /dev/null -w "%{http_code}\n" https://edition.cnn.com/politics  
200
```

6. Label the namespaces where the Istio components (the control plane and the gateways) run. If you deployed the Istio components to `istio-system`, the command is:

```
$ kubectl label namespace istio-system istio=system
```

## 7. Label the kube-system namespace.

```
$ kubectl label ns kube-system kube-system=true
```

## 8. Define a NetworkPolicy to limit the egress traffic from the test-egress namespace to traffic destined to istio-system, and to the kube-system DNS service (port 53):

```
$ cat <<EOF | kubectl apply -n test-egress -f -
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-egress-to-istio-system-and-kube-dns
spec:
  podSelector: {}
  policyTypes:
```

```
- Egress  
egress:  
- to:  
  - namespaceSelector:  
    matchLabels:  
      kube-system: "true"  
ports:  
- protocol: UDP  
  port: 53  
- to:  
  - namespaceSelector:  
    matchLabels:  
      istio: system
```

EOF

Network policies **are implemented by the network plugin in your Kubernetes**





cluster. Depending on your test cluster, the traffic may not be blocked in the following step.

9. Resend the previous HTTPS request to <https://edition.cnn.com/politics>. Now it should fail since the traffic is blocked by the network policy. Note that the `sleep` pod cannot bypass `istio-egressgateway`. The only way it can access `edition.cnn.com` is by using an Istio sidecar proxy and by directing the traffic to `istio-egressgateway`. This setting demonstrates that even if some malicious pod manages to bypass its sidecar

proxy, it will not be able to access external sites and will be blocked by the network policy.

```
$ kubectl exec "$(kubectl get pod -n test-egress -l app=sleep -o jsonpath={.items..metadata.name})" -n test-egress -c sleep -- curl -v -sS https://edition.cnn.com/politics  
Hostname was NOT found in DNS cache  
    Trying 151.101.65.67...  
    Trying 2a04:4e42:200::323...  
Immediate connect fail for 2a04:4e42:200::323: Cannot assign requested address  
    Trying 2a04:4e42:400::323...  
Immediate connect fail for 2a04:4e42:400::323: Cannot assign requested address  
    Trying 2a04:4e42:600::323...  
Immediate connect fail for 2a04:4e42:600::323: Cannot assign requested address  
    Trying 2a04:4e42::323...  
Immediate connect fail for 2a04:4e42::323: Cannot assign requested address  
connect to 151.101.65.67 port 443 failed: Connection timed out
```

- Now inject an Istio sidecar proxy into the sleep pod in the test-egress namespace by first enabling automatic sidecar proxy injection in the test-egress namespace:

```
$ kubectl label namespace test-egress istio-injection=enabled
```

- Then redeploy the sleep deployment:

```
$ kubectl delete deployment sleep -n test-egress
$ kubectl apply -f @samples/sleep/sleep.yaml@ -n test-egress
```

- Check that the deployed pod has two containers, including the Istio sidecar proxy (`istio-proxy`):

```
$ kubectl get pod "$(kubectl get pod -n test-egress -l app=sleep -o jsonpath={.items..metadata.name})" -n test-egress -o jsonpath='{.spec.containers[*].name}'  
sleep istio-proxy
```

3. Create the same destination rule as for the sleep pod in the default namespace to direct the traffic through the egress gateway:

```
$ kubectl apply -n test-egress -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: egressgateway-for-cnn
spec:
  host: istio-egressgateway.istio-system.svc.cluster.local
  subsets:
  - name: cnn
EOF
```

4. Send an HTTPS request to <https://edition.cnn.com/politics>. Now it should succeed since the traffic flows to `istio-egressgateway` in the `istio-system` namespace, which is allowed by the Network Policy you defined. `istio-egressgateway` forwards the traffic to

edition.cnn.com.

```
$ kubectl exec "$(kubectl get pod -n test-egress -l app=sleep -o jsonpath={.items..metadata.name})" -n test-egress -c sleep -- curl -sS -o /dev/null -w "%{http_code}\n" https://edition.cnn.com/politics  
200
```

5. Check the log of the egress gateway's proxy. If Istio is deployed in the `istio-system` namespace, the command to print the log is:

```
$ kubectl logs -l istio=egressgateway -n istio-system
```

You should see a line similar to the following:

```
[2020-03-06T18:12:33.101Z] " - - " 0 - "-" "-" 906 1352475  
35 - "-" "-" "-" "-" "151.101.193.67:443" outbound|443||edi  
tion.cnn.com 172.30.223.53:39460 172.30.223.53:443 172.30.2  
23.58:38138 edition.cnn.com -
```

# Cleanup network policies

1. Delete the resources created in this section:

```
$ kubectl delete -f @samples/sleep/sleep.yaml@ -n test-egress  
$ kubectl delete destinationrule egressgateway-for-cnn -n test-egress  
$ kubectl delete networkpolicy allow-egress-to-istio-system  
-and-kube-dns -n test-egress  
$ kubectl label namespace kube-system kube-system-  
$ kubectl label namespace istio-system istio-  
$ kubectl delete namespace test-egress
```

2. Follow the steps in the Cleanup HTTPS gateway section.

# Troubleshooting

1. If mutual TLS Authentication is enabled, verify the correct certificate of the egress gateway:

```
$ kubectl exec -i -n istio-system "$(kubectl get pod -l istio=egressgateway -n istio-system -o jsonpath='{.items[0].metadata.name}')" -- cat /etc/certs/cert-chain.pem | openssl x509 -text -noout | grep 'Subject Alternative Name' -A 1 X509v3 Subject Alternative Name:  
        URI:spiffe://cluster.local/ns/istio-system/sa/istio-egressgateway-service-account
```

2. For HTTPS traffic (TLS originated by the application), test the traffic flow by using the *openssl* command. *openssl* has an explicit option for setting the SNI, namely `-servername`.

```
$ kubectl exec "$SOURCE_POD" -c sleep -- openssl s_client -connect edition.cnn.com:443 -servername edition.cnn.com
CONNECTED(00000003)

...
Certificate chain
0 s:/C=US/ST=California/L=San Francisco/O=Fastly, Inc./CN=
turner-tls.map.fastly.net
    i:/C=BE/O=GlobalSign nv-sa/CN=GlobalSign CloudSSL CA - S
HA256 - G3
1 s:/C=BE/O=GlobalSign nv-sa/CN=GlobalSign CloudSSL CA - S
HA256 - G3
    i:/C=BE/O=GlobalSign nv-sa/OU=Root CA/CN=GlobalSign Root
CA
---
Server certificate
-----BEGIN CERTIFICATE-----
...
```

If you get the certificate as in the output above,

your traffic is routed correctly. Check the statistics of the egress gateway's proxy and see a counter that corresponds to your requests (sent by *openssl* and *curl*) to *edition.cnn.com*.

```
$ kubectl exec "$(kubectl get pod -l istio=egressgateway -n istio-system -o jsonpath='{.items[0].metadata.name}')" -c istio-proxy -n istio-system -- pilot-agent request GET stats | grep edition.cnn.com.upstream_cx_total
cluster.outbound|443|edition.cnn.com.upstream_cx_total: 2
```

## Cleanup

Shutdown the sleep service:

```
$ kubectl delete -f @samples/sleep/sleep.yaml@
```



# Egress Gateways with TLS Origination

⌚ 9 minute read ✓ page test

---

The TLS Origination for Egress Traffic example shows how to configure Istio to perform TLS origination for traffic to an external service. The Configure an Egress Gateway example shows how to configure Istio to direct egress traffic through a dedicated *egress*

*gateway* service. This example combines the previous two by describing how to configure an egress gateway to perform TLS origination for traffic to external services.

## Before you begin

- Setup Istio by following the instructions in the Installation guide.
- Start the sleep sample which will be used as a test source for external calls.

If you have enabled automatic sidecar injection, do

```
$ kubectl apply -f @samples/sleep/sleep.yaml@
```

otherwise, you have to manually inject the sidecar before deploying the `sleep` application:

```
$ kubectl apply -f <(istioctl kube-inject -f @samples/sleep/sleep.yaml@)
```

Note that any pod that you can `exec` and `curl` from would do.

- Create a shell variable to hold the name of the source pod for sending requests to external services. If you used the `sleep` sample, run:

```
$ export SOURCE_POD=$(kubectl get pod -l app=sleep -o jsonpath={.items..metadata.name})
```

- For macOS users, verify that you are using `openssl` version 1.1 or later:

```
$ openssl version -a | grep OpenSSL  
OpenSSL 1.1.1g  21 Apr 2020
```

If the previous command outputs a version 1.1 or later, as shown, your `openssl` command should work correctly with the instructions in this task. Otherwise, upgrade your `openssl` or try a different implementation of `openssl`, for example on a Linux machine.

- Deploy Istio egress gateway.
- Enable Envoy's access logging

## **Perform TLS origination with an egress gateway**

This section describes how to perform the same TLS origination as in the TLS Origination for Egress Traffic example, only this time using an egress gateway. Note that in this case the TLS origination will be done

by the egress gateway, as opposed to by the sidecar in the previous example.

1. Define a ServiceEntry for edition.cnn.com:

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: cnn
spec:
  hosts:
  - edition.cnn.com
  ports:
  - number: 80
    name: http
    protocol: HTTP
  - number: 443
    name: https
    protocol: HTTPS
  resolution: DNS
EOF
```

2. Verify that your ServiceEntry was applied correctly

by sending a request to  
`http://edition.cnn.com/politics.`

```
$ kubectl exec "${SOURCE_POD}" -c sleep -- curl -sSL -o /dev/null -D - http://edition.cnn.com/politics
HTTP/1.1 301 Moved Permanently
...
location: https://edition.cnn.com/politics
...
```

Your ServiceEntry was configured correctly if you see *301 Moved Permanently* in the output.

3. Create an egress Gateway for *edition.cnn.com*, port 80, and a destination rule for sidecar requests that will be directed to the egress gateway.

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: istio-egressgateway
spec:
  selector:
    istio: egressgateway
  servers:
  - port:
      number: 80
      name: https-port-for-tls-origination
      protocol: HTTPS
    hosts:
    - edition.cnn.com
    tls:
      mode: ISTIO_MUTUAL
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
```

```
metadata:
  name: egressgateway-for-cnn
spec:
  host: istio-egressgateway.istio-system.svc.cluster.local
  subsets:
    - name: cnn
      trafficPolicy:
        loadBalancer:
          simple: ROUND_ROBIN
      portLevelSettings:
        - port:
            number: 80
            tls:
              mode: ISTIO_MUTUAL
              sni: edition.cnn.com
```

EOF

4. Define a VirtualService to direct the traffic through the egress gateway, and a DestinationRule

to perform TLS origination for requests to edition.cnn.com:

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: direct-cnn-through-egress-gateway
spec:
  hosts:
  - edition.cnn.com
  gateways:
  - istio-egressgateway
  - mesh
  http:
  - match:
    - gateways:
      - mesh
  port: 80
```

```
route:
  - destination:
      host: istio-egressgateway.istio-system.svc.cluster.local
        subset: cnn
        port:
          number: 80
          weight: 100
  - match:
    - gateways:
      - istio-egressgateway
        port: 80
    route:
      - destination:
          host: edition.cnn.com
          port:
            number: 443
            weight: 100
---
apiVersion: networking.istio.io/v1alpha3
```

```
kind: DestinationRule
metadata:
  name: originate-tls-for-edition-cnn-com
spec:
  host: edition.cnn.com
  trafficPolicy:
    loadBalancer:
      simple: ROUND_ROBIN
    portLevelSettings:
      - port:
          number: 443
        tls:
          mode: SIMPLE # initiates HTTPS for connections to edition.cnn.com
EOF
```

## 5. Send an HTTP request to [http://edition.cnn.com/politics.](http://edition.cnn.com/politics)

```
$ kubectl exec "${SOURCE_POD}" -c sleep -- curl -sSL -o /dev/null -D - http://edition.cnn.com/politics  
HTTP/1.1 200 OK  
...
```

The output should be the same as in the TLS Origination for Egress Traffic example, with TLS origination: without the *301 Moved Permanently* message.

6. Check the log of the `istio-egressgateway` pod and you should see a line corresponding to our request. If Istio is deployed in the `istio-system` namespace, the command to print the log is:

```
$ kubectl logs -l istio=egressgateway -c istio-proxy -n istio-system | tail
```

You should see a line similar to the following:

```
[2020-06-30T16:17:56.763Z] "GET /politics HTTP/2" 200 - "-"  
"-" 0 1295938 529 89 "10.244.0.171" "curl/7.64.0" "cf76518  
d-3209-9ab7-a1d0-e6002728ef5b" "edition.cnn.com" "151.101.1  
29.67:443" outbound|443||edition.cnn.com 10.244.0.170:54280  
10.244.0.170:8080 10.244.0.171:35628 - -
```

# Cleanup the TLS origination example

Remove the Istio configuration items you created:

```
$ kubectl delete gateway istio-egressgateway  
$ kubectl delete serviceentry cnn  
$ kubectl delete virtualservice direct-cnn-through-egress-gatewa  
y  
$ kubectl delete destinationrule originate-tls-for-edition-cnn-c  
om  
$ kubectl delete destinationrule egressgateway-for-cnn
```

**Perform mutual TLS  
origination with an egress  
gateway**

Similar to the previous section, this section describes how to configure an egress gateway to perform TLS origination for an external service, only this time using a service that requires mutual TLS.

This example is considerably more involved because you need to first:

1. generate client and server certificates
2. deploy an external service that supports the mutual TLS protocol
3. redeploy the egress gateway with the needed mutual TLS certs

Only then can you configure the external traffic to go through the egress gateway which will perform TLS origination.

## **Generate client and server certificates and keys**

For this task you can use your favorite tool to generate certificates and keys. The commands below use openssl

1. Create a root certificate and private key to sign

the certificate for your services:

```
$ openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 -subj '/O=example Inc./CN=example.com' -keyout example.com.key -out example.com.crt
```

## 2. Create a certificate and a private key for my-nginx.mesh-external.svc.cluster.local:

```
$ openssl req -out my-nginx.mesh-external.svc.cluster.local.csr -newkey rsa:2048 -nodes -keyout my-nginx.mesh-external.svc.cluster.local.key -subj "/CN=my-nginx.mesh-external.svc.cluster.local/O=some organization"  
$ openssl x509 -req -days 365 -CA example.com.crt -CAkey example.com.key -set_serial 0 -in my-nginx.mesh-external.svc.cluster.local.csr -out my-nginx.mesh-external.svc.cluster.local.crt
```

### 3. Generate client certificate and private key:

```
$ openssl req -out client.example.com.csr -newkey rsa:2048  
-nodes -keyout client.example.com.key -subj "/CN=client.exa  
mple.com/O=client organization"  
$ openssl x509 -req -days 365 -CA example.com.crt -CAkey ex  
ample.com.key -set_serial 1 -in client.example.com.csr -out  
client.example.com.crt
```

## Deploy a mutual TLS server

To simulate an actual external service that supports the mutual TLS protocol, deploy an NGINX server in your Kubernetes cluster, but running outside of the

Istio service mesh, i.e., in a namespace without Istio sidecar proxy injection enabled.

1. Create a namespace to represent services outside the Istio mesh, namely `mesh-external`. Note that the sidecar proxy will not be automatically injected into the pods in this namespace since the automatic sidecar injection was not enabled on it.

```
$ kubectl create namespace mesh-external
```

2. Create Kubernetes Secrets to hold the server's and CA certificates.

```
$ kubectl create -n mesh-external secret tls nginx-server-certs --key my-nginx.mesh-external.svc.cluster.local.key --cert my-nginx.mesh-external.svc.cluster.local.crt
$ kubectl create -n mesh-external secret generic nginx-ca-cert --from-file=example.com.crt
```

### 3. Create a configuration file for the NGINX server:

```
$ cat <<\EOF > ./nginx.conf
events {
}

http {
    log_format main '$remote_addr - $remote_user [$time_local]
] $status '
    '"$request" $body_bytes_sent "$http_referer" '
    '"$http_user_agent" "$http_x_forwarded_for"';
    access_log /var/log/nginx/access.log main;
    error_log  /var/log/nginx/error.log;
```

```
server {  
    listen 443 ssl;  
  
    root /usr/share/nginx/html;  
    index index.html;  
  
    server_name my-nginx.mesh-external.svc.cluster.local;  
    ssl_certificate /etc/nginx-server-certs/tls.crt;  
    ssl_certificate_key /etc/nginx-server-certs/tls.key;  
    ssl_client_certificate /etc/nginx-ca-certs/example.com.  
crt;  
    ssl_verify_client on;  
}  
}  
EOF
```

4. Create a Kubernetes ConfigMap to hold the configuration of the NGINX server:

```
$ kubectl create configmap nginx-configmap -n mesh-external  
--from-file=nginx.conf=./nginx.conf
```

## 5. Deploy the NGINX server:

```
$ kubectl apply -f - <<EOF  
apiVersion: v1  
kind: Service  
metadata:  
  name: my-nginx  
  namespace: mesh-external  
  labels:  
    run: my-nginx  
spec:  
  ports:  
  - port: 443  
    protocol: TCP  
  selector:  
    run: my-nginx
```

```
---
```

```
apiVersion: apps/v1
```

```
kind: Deployment
```

```
metadata:
```

```
    name: my-nginx
```

```
    namespace: mesh-external
```

```
spec:
```

```
    selector:
```

```
        matchLabels:
```

```
            run: my-nginx
```

```
replicas: 1
```

```
template:
```

```
    metadata:
```

```
        labels:
```

```
            run: my-nginx
```

```
spec:
```

```
    containers:
```

```
        - name: my-nginx
```

```
            image: nginx
```

```
            ports:
```

```
- containerPort: 443
volumeMounts:
  - name: nginx-config
    mountPath: /etc/nginx
    readOnly: true
  - name: nginx-server-certs
    mountPath: /etc/nginx-server-certs
    readOnly: true
  - name: nginx-ca-certs
    mountPath: /etc/nginx-ca-certs
    readOnly: true
volumes:
  - name: nginx-config
    configMap:
      name: nginx-configmap
  - name: nginx-server-certs
    secret:
      secretName: nginx-server-certs
  - name: nginx-ca-certs
    secret:
```

EOF

secretName: nginx-ca-certs

# Configure mutual TLS origination for egress traffic

1. Create Kubernetes Secrets to hold the client's certificates:

```
$ kubectl create secret -n istio-system generic client-credential --from-file=tls.key=client.example.com.key \
--from-file=tls.crt=client.example.com.crt --from-file=ca.crt=example.com.crt
```

The secret **must** be created in the same namespace as the egress gateway is deployed in, `istio-system` in this case.

To support integration with various tools, Istio supports a few different Secret formats.

In this example, a single generic Secret with keys `tls.key`, `tls.crt`, and `ca.crt` is used.

2. Create an egress Gateway for `my-nginx.mesh-external.svc.cluster.local`, port 443, and

destination rules and virtual services to direct the traffic through the egress gateway and from the egress gateway to the external service.

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: istio-egressgateway
spec:
  selector:
    istio: egressgateway
  servers:
  - port:
      number: 443
      name: https
      protocol: HTTPS
    hosts:
    - my-nginx.mesh-external.svc.cluster.local
```

```
    tls:
      mode: ISTIO_MUTUAL
    ---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: egressgateway-for-nginx
spec:
  host: istio-egressgateway.istio-system.svc.cluster.local
  subsets:
  - name: nginx
    trafficPolicy:
      loadBalancer:
        simple: ROUND_ROBIN
      portLevelSettings:
      - port:
          number: 443
        tls:
          mode: ISTIO_MUTUAL
          sni: my-nginx.mesh-external.svc.cluster.local
```

### 3. Define a `VirtualService` to direct the traffic through the egress gateway:

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: direct-nginx-through-egress-gateway
spec:
  hosts:
  - my-nginx.mesh-external.svc.cluster.local
  gateways:
  - istio-egressgateway
  - mesh
  http:
  - match:
    - gateways:
```

```
    - mesh
    port: 80
  route:
    - destination:
        host: istio-egressgateway.istio-system.svc.cluster.local
      local
        subset: nginx
        port:
          number: 443
          weight: 100
    - match:
        - gateways:
            - istio-egressgateway
          port: 443
        route:
          - destination:
              host: my-nginx.mesh-external.svc.cluster.local
              port:
                number: 443
              weight: 100
```

4. Add a DestinationRule to perform mutual TLS origination

```
$ kubectl apply -n istio-system -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: originate-mtls-for-nginx
spec:
  host: my-nginx.mesh-external.svc.cluster.local
  trafficPolicy:
    loadBalancer:
      simple: ROUND_ROBIN
    portLevelSettings:
      - port:
          number: 443
        tls:
          mode: MUTUAL
          credentialName: client-credential # this must match
          the secret created earlier to hold client certs
          sni: my-nginx.mesh-external.svc.cluster.local
EOF
```

5. Send an HTTP request to `http://my-nginx.mesh-external.svc.cluster.local`:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -o jsonpath={.items..metadata.name})" -c sleep -- curl -ss http://my-nginx.mesh-external.svc.cluster.local
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...

```

6. Check the log of the `istio-egressgateway` pod for a line corresponding to our request. If Istio is deployed in the `istio-system` namespace, the command to print the log is:

```
$ kubectl logs -l istio=egressgateway -n istio-system | grep 'my-nginx.mesh-external.svc.cluster.local' | grep HTTP
```

You should see a line similar to the following:

```
[2018-08-19T18:20:40.096Z] "GET / HTTP/1.1" 200 - 0 612 7 5  
"172.30.146.114" "curl/7.35.0" "b942b587-fac2-9756-8ec6-30  
3561356204" "my-nginx.mesh-external.svc.cluster.local" "172  
.21.72.197:443"
```

# Cleanup the mutual TLS origination example

## 1. Remove created Kubernetes resources:

```
$ kubectl delete secret nginx-server-certs nginx-ca-certs -n mesh-external  
$ kubectl delete secret client-credential istio-egressgateway-certs istio-egressgateway-ca-certs nginx-client-certs nginx-ca-certs -n istio-system  
$ kubectl delete configmap nginx-configmap -n mesh-external  
$ kubectl delete service my-nginx -n mesh-external  
$ kubectl delete deployment my-nginx -n mesh-external  
$ kubectl delete namespace mesh-external  
$ kubectl delete gateway istio-egressgateway  
$ kubectl delete virtualservice direct-nginx-through-egress-gateway  
$ kubectl delete destinationrule -n istio-system originate-mtls-for-nginx  
$ kubectl delete destinationrule egressgateway-for-nginx
```

## 2. Delete the certificates and private keys:

```
$ rm example.com.crt example.com.key my-nginx.mesh-external  
.svc.cluster.local.crt my-nginx.mesh-external.svc.cluster.l  
ocal.key my-nginx.mesh-external.svc.cluster.local.csr clien  
t.example.com.crt client.example.com.csr client.example.com  
.key
```

3. Delete the generated configuration files used in this example:

```
$ rm ./nginx.conf  
$ rm ./gateway-patch.json
```

# Cleanup

## Delete the sleep service and deployment:

```
$ kubectl delete service sleep  
$ kubectl delete deployment sleep
```

# Egress using Wildcard Hosts

⌚ 12 minute read ✓ page test

---

The Accessing External Services task and the Configure an Egress Gateway example describe how to configure egress traffic for specific hostnames, like edition.cnn.com. This example shows how to enable egress traffic for a set of hosts in a common domain,

for example \*.wikipedia.org, instead of configuring each and every host separately.

## Background

Suppose you want to enable egress traffic in Istio for the wikipedia.org sites in all languages. Each version of wikipedia.org in a particular language has its own hostname, e.g., en.wikipedia.org and de.wikipedia.org in the English and the German languages, respectively. You want to enable egress traffic by common

configuration items for all the Wikipedia sites, without the need to specify every language's site separately.

## Before you begin

- Install Istio using the `demo` configuration profile **and** with the blocking-by-default outbound traffic policy:

```
$ istioctl install --set profile=demo --set meshConfig.outboundTrafficPolicy.mode=REGISTRY_ONLY
```



You can run this task on an Istio configuration other than the `demo` profile as long as you make sure to deploy the Istio egress gateway, enable Envoy's access logging, and apply the blocking-by-default outbound traffic policy in your installation. You will also need to add the second gateway using your own `IstioOperator` CR instead of the one shown in `setup-egress-gateway-with-SNI-proxy`.

- Deploy the sleep sample app to use as a test source for sending requests. If you have automatic sidecar injection enabled, run the following command to deploy the sample app:

```
$ kubectl apply -f @samples/sleep/sleep.yaml@
```

Otherwise, manually inject the sidecar before deploying the sleep application with the following command:

```
$ kubectl apply -f <(istioctl kube-inject -f @samples/sleep/sleep.yaml@)
```



You can use any pod with `curl` installed as a test source.

- Set the `SOURCE_POD` environment variable to the name of your source pod:

```
$ export SOURCE_POD=$(kubectl get pod -l app=sleep -o jsonpath={.items..metadata.name})
```

# Configure direct traffic to a

# wildcard host

The first, and simplest, way to access a set of hosts within a common domain is by configuring a simple `ServiceEntry` with a wildcard host and calling the services directly from the sidecar. When calling services directly (i.e., not via an egress gateway), the configuration for a wildcard host is no different than that of any other (e.g., fully qualified) host, only much more convenient when there are many hosts within the common domain.

 Note that the configuration below can be easily bypassed by a malicious application. For a secure egress traffic control, direct the traffic through an egress gateway.

 Note that the `DNS` resolution cannot be used for wildcard hosts. This is why the `NONE` resolution (omitted since it is the default) is used in the service entry below.

## 1. Define a ServiceEntry for \*.wikipedia.org:

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: wikipedia
spec:
  hosts:
  - "*.wikipedia.org"
  ports:
  - number: 443
    name: https
    protocol: HTTPS
EOF
```

## 2. Send HTTPS requests to <https://en.wikipedia.org> and <https://de.wikipedia.org>:

```
$ kubectl exec "$SOURCE_POD" -c sleep -- sh -c 'curl -s https://en.wikipedia.org/wiki/Main_Page | grep -o "<title>.*</title>"; curl -s https://de.wikipedia.org/wiki/Wikipedia:Hauptseite | grep -o "<title>.*</title>"'  
<title>Wikipedia, the free encyclopedia</title>  
<title>Wikipedia – Die freie Enzyklopädie</title>
```

# Cleanup direct traffic to a wildcard host

```
$ kubectl delete serviceentry wikipedia
```

# Configure egress gateway traffic to a wildcard host

The configuration for accessing a wildcard host via an egress gateway depends on whether or not the set of wildcard domains are served by a single common host. This is the case for `*.wikipedia.org`. All of the language-specific sites are served by every one of the `wikipedia.org` servers. You can route the traffic to an IP of any `*.wikipedia.org` site, including `www.wikipedia.org`, and it will manage to serve any specific site.

In the general case, where all the domain names of a wildcard are not served by a single hosting server, a more complex configuration is required.

## **Wildcard configuration for a single hosting server**

When all wildcard hosts are served by a single server, the configuration for egress gateway-based access to a wildcard host is very similar to that of any host, with one exception: the configured route destination will not be the same as the configured host, i.e., the

wildcard. It will instead be configured with the host of the single server for the set of domains.

1. Create an egress Gateway for `*.wikipedia.org`, a destination rule and a virtual service to direct the traffic through the egress gateway and from the egress gateway to the external service.

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: istio-egressgateway
spec:
  selector:
    istio: egressgateway
  servers:
```

```
- port:  
    number: 443  
    name: https  
    protocol: HTTPS  
hosts:  
- "*.wikipedia.org"  
tls:  
mode: PASSTHROUGH  
---  
apiVersion: networking.istio.io/v1alpha3  
kind: DestinationRule  
metadata:  
  name: egressgateway-for-wikipedia  
spec:  
  host: istio-egressgateway.istio-system.svc.cluster.local  
  subsets:  
  - name: wikipedia  
---  
apiVersion: networking.istio.io/v1alpha3  
kind: VirtualService
```

```
metadata:
  name: direct-wikipedia-through-egress-gateway
spec:
  hosts:
    - "*.wikipedia.org"
  gateways:
    - mesh
    - istio-egressgateway
  tls:
    - match:
        - gateways:
            - mesh
        port: 443
        sniHosts:
          - "*.wikipedia.org"
  route:
    - destination:
        host: istio-egressgateway.istio-system.svc.cluster.local
          subset: wikipedia
```

```
    port:  
        number: 443  
        weight: 100  
    - match:  
        - gateways:  
            - istio-egressgateway  
        port: 443  
        sniHosts:  
            - "*.wikipedia.org"  
    route:  
        - destination:  
            host: www.wikipedia.org  
            port:  
                number: 443  
            weight: 100
```

EOF

2. Create a ServiceEntry for the destination server, *www.wikipedia.org*.

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: www-wikipedia
spec:
  hosts:
  - www.wikipedia.org
  ports:
  - number: 443
    name: https
    protocol: HTTPS
  resolution: DNS
EOF
```

3. Send HTTPS requests to <https://en.wikipedia.org> and <https://de.wikipedia.org>:

```
$ kubectl exec "$SOURCE_POD" -c sleep -- sh -c 'curl -s https://en.wikipedia.org/wiki/Main_Page | grep -o "<title>.*</title>"; curl -s https://de.wikipedia.org/wiki/Wikipedia:Hauptseite | grep -o "<title>.*</title>"'
<title>Wikipedia, the free encyclopedia</title>
<title>Wikipedia – Die freie Enzyklopädie</title>
```

4. Check the statistics of the egress gateway's proxy for the counter that corresponds to your requests to `*.wikipedia.org`. If Istio is deployed in the `istio-system` namespace, the command to print the counter is:

```
$ kubectl exec "$(kubectl get pod -l istio=egressgateway -n istio-system -o jsonpath='{.items[0].metadata.name}')" -c istio-proxy -n istio-system -- pilot-agent request GET clusters | grep '^outbound|443||www.wikipedia.org.*cx_total:' outbound|443|www.wikipedia.org::208.80.154.224:443::cx_total::2
```

Cleanup wildcard  
configuration for a single  
hosting server

```
$ kubectl delete serviceentry www-wikipedia
$ kubectl delete gateway istio-egressgateway
$ kubectl delete virtualservice direct-wikipedia-through-egress-
gateway
$ kubectl delete destinationrule egressgateway-for-wikipedia
```

## Wildcard configuration for arbitrary domains

The configuration in the previous section worked because all the `*.wikipedia.org` sites can be served by any one of the `wikipedia.org` servers. However, this is not always the case. For example, you may want to

configure egress control for access to more general wildcard domains like \*.com or \*.org.

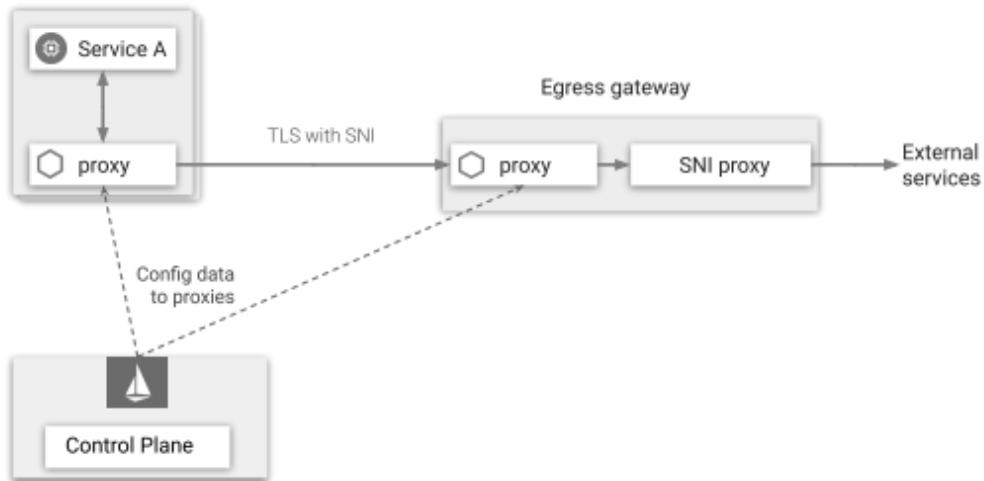
Configuring traffic to arbitrary wildcard domains introduces a challenge for Istio gateways. In the previous section you directed the traffic to [www.wikipedia.org](http://www.wikipedia.org), which was made known to your gateway during configuration. The gateway, however, would not know the IP address of any arbitrary host it receives in a request. This is due to a limitation of Envoy, the proxy used by the default Istio egress gateway. Envoy routes traffic either to predefined hosts, predefined IP addresses, or to the original destination IP address of the request. In the gateway

case, the original destination IP of the request is lost since the request is first routed to the egress gateway and its destination IP address is the IP address of the gateway.

Consequently, the Istio gateway based on Envoy cannot route traffic to an arbitrary host that is not preconfigured, and therefore is unable to perform traffic control for arbitrary wildcard domains. To enable such traffic control for HTTPS, and for any TLS, you need to deploy an SNI forward proxy in addition to Envoy. Envoy will route the requests destined for a wildcard domain to the SNI forward proxy, which, in turn, will forward the requests to the

destination specified by the SNI value.

The egress gateway with SNI proxy and the related parts of the Istio architecture are shown in the following diagram:



## Egress Gateway with SNI proxy

The following sections show you how to redeploy the egress gateway with an SNI proxy and then configure Istio to route HTTPS traffic through the gateway to arbitrary wildcard domains.

### Setup egress gateway with SNI proxy

In this section you deploy an egress gateway with an

SNI proxy in addition to the standard Istio Envoy proxy. This example uses Nginx for the SNI proxy, although any SNI proxy that is capable of routing traffic according to arbitrary, not-preconfigured, SNI values would do. The SNI proxy will listen on port 8443, although you can use any port other than the ports specified for the egress `Gateway` and for the `VirtualServices` bound to it. The SNI proxy will forward the traffic to port 443.

1. Create a configuration file for the Nginx SNI proxy. You may want to edit the file to specify additional Nginx settings, if required. Note that the `listen` directive of the `server` specifies port

8443, its proxy\_pass directive uses  
ssl\_preread\_server\_name with port 443 and  
ssl\_preread is on to enable SNI reading.

```
$ cat <<EOF > ./sni-proxy.conf
# setup custom path that do not require root access
pid /tmp/nginx.pid;

events {
}

stream {
    log_format log_stream '\$remote_addr [\$time_local] \$proto
    [\$ssl_preread_server_name]'
    '\$status \$bytes_sent \$bytes_received \$session_time';

    access_log /var/log/nginx/access.log log_stream;
    error_log  /var/log/nginx/error.log;
```

```
# tcp forward proxy by SNI
server {
    resolver 8.8.8.8 ipv6=off;
    listen      127.0.0.1:18443;
    proxy_pass  \$ssl_preread_server_name:443;
    ssl_preread on;
}
}
EOF
```

2. Create a Kubernetes ConfigMap to hold the configuration of the Nginx SNI proxy:

```
$ kubectl create configmap egress-sni-proxy-configmap -n istio-system --from-file=nginx.conf=./sni-proxy.conf
```

3. Create an IstioOperator CR to add a new egress gateway with SNI proxy:

```
$ istioctl manifest generate -f - <<EOF > ./egressgateway-with-sni-proxy.yaml
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  # Only generate a gateway component defined below.
  # Using this with "istioctl install" will reconcile and remove existing control-plane components.
  # Instead use "istioctl manifest generate" or "kubectl create" if using the istio operator.
  profile: empty
  components:
    egressGateways:
      - name: istio-egressgateway-with-sni-proxy
        enabled: true
        label:
          app: istio-egressgateway-with-sni-proxy
          istio: egressgateway-with-sni-proxy
    k8s:
      service:
```

```
ports:
  - port: 443
    targetPort: 8443
    name: https
overlays:
  - kind: Deployment
    name: istio-egressgateway-with-sni-proxy
patches:
  - path: spec.template.spec.containers[-1]
    value: |
      name: sni-proxy
      image: nginx
      volumeMounts:
        - name: sni-proxy-config
          mountPath: /etc/nginx
          readOnly: true
      securityContext:
        runAsNonRoot: true
        runAsUser: 101
    - path: spec.template.spec.volumes[-1]
```

```
value: |
  name: sni-proxy-config
  configMap:
    name: egress-sni-proxy-configmap
    defaultMode: 292 # 0444
EOF
```

#### 4. Deploy the new gateway:

```
$ kubectl apply -f ./egressgateway-with-sni-proxy.yaml
```

#### 5. Verify that the new egress gateway is running.

Note that the pod has two containers (one is the Envoy proxy and the second one is the SNI proxy).

```
$ kubectl get pod -l istio=egressgateway-with-sni-proxy -n istio-system
NAME                                     READY
STATUS      RESTARTS   AGE
istio-egressgateway-with-sni-proxy-79f6744569-pf9t2   2/2
    Running   0          17s
```

6. Create a service entry with a static address equal to 127.0.0.1 (`localhost`), and disable mutual TLS for traffic directed to the new service entry:

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: sni-proxy
spec:
  hosts:
```

```
- sni-proxy.local
location: MESH_EXTERNAL
ports:
- number: 18443
  name: tcp
  protocol: TCP
resolution: STATIC
endpoints:
- address: 127.0.0.1
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: disable-mtls-for-sni-proxy
spec:
  host: sni-proxy.local
  trafficPolicy:
    tls:
      mode: DISABLE
```

EOF

# Configure traffic through egress gateway with SNI proxy

1. Define a ServiceEntry for \*.wikipedia.org:

```
$ cat <<EOF | kubectl create -f -
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: wikipedia
spec:
  hosts:
  - "*.wikipedia.org"
  ports:
  - number: 443
    name: tls
    protocol: TLS
EOF
```

2. Create an egress Gateway for *\*.wikipedia.org*, port 443, protocol TLS, and a virtual service to direct the traffic destined for *\*.wikipedia.org* through the gateway.

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: istio-egressgateway-with-sni-proxy
spec:
  selector:
    istio: egressgateway-with-sni-proxy
  servers:
  - port:
      number: 443
      name: tls-egress
      protocol: TLS
    hosts:
    - "* wikipedia.org"
    tls:
      mode: ISTIO_MUTUAL
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
```

```
metadata:
  name: egressgateway-for-wikipedia
spec:
  host: istio-egressgateway-with-sni-proxy.istio-system.svc
.cluster.local
  subsets:
    - name: wikipedia
      trafficPolicy:
        loadBalancer:
          simple: ROUND_ROBIN
      portLevelSettings:
        - port:
            number: 443
            tls:
              mode: ISTIO_MUTUAL
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: direct-wikipedia-through-egress-gateway
```

```
spec:
  hosts:
    - "* wikipedia.org"
  gateways:
    - mesh
    - istio-egressgateway-with-sni-proxy
  tls:
    - match:
        - gateways:
            - mesh
        port: 443
        sniHosts:
          - "* wikipedia.org"
  route:
    - destination:
        host: istio-egressgateway-with-sni-proxy.istio-system.svc.cluster.local
        subset: wikipedia
        port:
          number: 443
```

```
    weight: 100
  tcp:
    - match:
      - gateways:
        - istio-egressgateway-with-sni-proxy
        port: 443
      route:
        - destination:
          host: sni-proxy.local
          port:
            number: 18443
        weight: 100
  ---
# The following filter is used to forward the original SNI
# (sent by the application) as the SNI of the
# mutual TLS connection.
# The forwarded SNI will be will be used to enforce policies
# based on the original SNI value.
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
```

```
metadata:  
  name: forward-downstream-sni  
spec:  
  configPatches:  
    - applyTo: NETWORK_FILTER  
      match:  
        context: SIDECAR_OUTBOUND  
      listener:  
        portNumber: 443  
      filterChain:  
        filter:  
          name: istio.stats  
patch:  
  operation: INSERT_BEFORE  
  value:  
    name: forward_downstream_sni  
    config: {}  
EOF
```

3. Add an EnvoyFilter to the gateway, to prevent it

# from being deceived.

```
$ kubectl apply -n istio-system -f - <<EOF
# The following filter verifies that the SNI of the mutual
TLS connection is
# identical to the original SNI issued by the client (the S
NI used for routing by the SNI proxy).
# The filter prevents the gateway from being deceived by a
malicious client: routing to one SNI while
# reporting some other value of SNI. If the original SNI do
es not match the SNI of the mutual TLS connection,
# the filter will block the connection to the external serv
ice.

apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: egress-gateway-sni-verifier
spec:
  workloadSelector:
    labels:
```

```
        app: istio-egressgateway-with-sni-proxy
configPatches:
- applyTo: NETWORK_FILTER
  match:
    context: GATEWAY
  listener:
    portNumber: 443
    filterChain:
      filter:
        name: istio.stats
  patch:
    operation: INSERT_BEFORE
    value:
      name: sni_verifier
      config: {}
EOF
```

4. Send HTTPS requests to <https://en.wikipedia.org> and <https://de.wikipedia.org>:

```
$ kubectl exec "$SOURCE_POD" -c sleep -- sh -c 'curl -s https://en.wikipedia.org/wiki/Main_Page | grep -o "<title>.*</title>"; curl -s https://de.wikipedia.org/wiki/Wikipedia:Hauptseite | grep -o "<title>.*</title>"'
<title>Wikipedia, the free encyclopedia</title>
<title>Wikipedia – Die freie Enzyklopädie</title>
```

5. Check the log of the egress gateway's Envoy proxy. If Istio is deployed in the `istio-system` namespace, the command to print the log is:

```
$ kubectl logs -l istio=egressgateway-with-sni-proxy -c istio-proxy -n istio-system
```

You should see lines similar to the following:

```
[2019-01-02T16:34:23.312Z] " - - - " 0 - 578 79141 624 - "-"
"- - -" "127.0.0.1:18443" outbound|18443||sni-proxy.loc
al 127.0.0.1:55018 172.30.109.84:443 172.30.109.112:45346 e
n.wikipedia.org
[2019-01-02T16:34:24.079Z] " - - - " 0 - 586 65770 638 - "-"
"- - -" "127.0.0.1:18443" outbound|18443||sni-proxy.loc
al 127.0.0.1:55034 172.30.109.84:443 172.30.109.112:45362 d
e.wikipedia.org
```

6. Check the logs of the SNI proxy. If Istio is deployed in the `istio-system` namespace, the command to print the log is:

```
$ kubectl logs -l istio=egressgateway-with-sni-proxy -n istio-system -c sni-proxy
127.0.0.1 [01/Aug/2018:15:32:02 +0000] TCP [en.wikipedia.org]200 81513 280 0.600
127.0.0.1 [01/Aug/2018:15:32:03 +0000] TCP [de.wikipedia.org]200 67745 291 0.659
```

## Cleanup wildcard configuration for arbitrary domains

1. Delete the configuration items for *\*.wikipedia.org*:

```
$ kubectl delete serviceentry wikipedia
$ kubectl delete gateway istio-egressgateway-with-sni-proxy
$ kubectl delete virtualservice direct-wikipedia-through-egress-gateway
$ kubectl delete destinationrule egressgateway-for-wikipedia
$ kubectl delete --ignore-not-found=true envoyfilter forward-downstream-sni
$ kubectl delete --ignore-not-found=true envoyfilter -n istio-system egress-gateway-sni-verifier
```

## 2. Delete the configuration items for the egressgateway-with-sni-proxy deployment:

```
$ kubectl delete serviceentry sni-proxy  
$ kubectl delete destinationrule disable-mtls-for-sni-proxy  
$ kubectl delete configmap egress-sni-proxy-configmap -n istio-system  
$ kubectl delete -f ./egressgateway-with-sni-proxy.yaml
```

### 3. Remove the configuration files you created:

```
$ rm ./sni-proxy.conf ./egressgateway-with-sni-proxy.yaml
```

## Cleanup

- Shutdown the sleep service:

```
$ kubectl delete -f @samples/sleep/sleep.yaml@
```

- Uninstall Istio from your cluster:

```
$ istioctl x uninstall --purge
```

# Kubernetes Services for Egress Traffic

⌚ 6 minute read ✓ page test

---

Kubernetes ExternalName services and Kubernetes services with Endpoints let you create a local DNS *alias* to an external service. This DNS alias has the same form as the DNS entries for local services, namely <service name>.<namespace>

`name>.svc.cluster.local`. DNS aliases provide *location transparency* for your workloads: the workloads can call local and external services in the same way. If at some point in time you decide to deploy the external service inside your cluster, you can just update its Kubernetes service to reference the local version. The workloads will continue to operate without any change.

This task shows that these Kubernetes mechanisms for accessing external services continue to work with Istio. The only configuration step you must perform is to use a TLS mode other than Istio's mutual TLS. The external services are not part of an Istio service mesh

so they cannot perform the mutual TLS of Istio. You must set the TLS mode according to the TLS requirements of the external service and according to the way your workload accesses the external service. If your workload issues plain HTTP requests and the external service requires TLS, you may want to perform TLS origination by Istio. If your workload already uses TLS, the traffic is already encrypted and you can just disable Istio's mutual TLS.

This page describes how Istio can integrate with existing Kubernetes configurations. For

new deployments, we recommend following Accessing Egress Services.

While the examples in this task use HTTP protocols, Kubernetes Services for egress traffic work with other protocols as well.

## Before you begin

- Setup Istio by following the instructions in the

## Installation guide.



The egress gateway and access logging will be enabled if you install the demo configuration profile.

- Deploy the sleep sample app to use as a test source for sending requests. If you have automatic sidecar injection enabled, run the following command to deploy the sample app:

```
$ kubectl apply -f @samples/sleep/sleep.yaml@
```

Otherwise, manually inject the sidecar before deploying the sleep application with the following command:

```
$ kubectl apply -f <(istioctl kube-inject -f @samples/sleep/sleep.yaml@)
```



You can use any pod with `curl` installed as a test source.

- Set the `SOURCE_POD` environment variable to the name of your source pod:

```
$ export SOURCE_POD=$(kubectl get pod -l app=sleep -o jsonpath={.items..metadata.name})
```

- Create a namespace for a source pod without Istio control:

```
$ kubectl create namespace without-istio
```

- Start the sleep sample in the without-istio namespace.

```
$ kubectl apply -f @samples/sleep/sleep.yaml@ -n without-istio
```

- To send requests, create the SOURCE\_POD\_WITHOUT\_ISTIO environment variable to

store the name of the source pod:

```
$ export SOURCE_POD_WITHOUT_ISTIO=$(kubectl get pod -n without-istio -l app=sleep -o jsonpath={.items..metadata.name})"
```

- Verify that the Istio sidecar was not injected, that is the pod has one container:

```
$ kubectl get pod "$SOURCE_POD_WITHOUT_ISTIO" -n without-istio
```

NAME	READY	STATUS	RESTARTS	AGE
sleep-66c8d79ff5-8tqrl	1/1	Running	0	32s

# Kubernetes ExternalName service to access an external service

1. Create a Kubernetes ExternalName service for `httpbin.org` in the default namespace:

```
$ kubectl apply -f - <<EOF
kind: Service
apiVersion: v1
metadata:
  name: my-httpbin
spec:
  type: ExternalName
  externalName: httpbin.org
  ports:
  - name: http
    protocol: TCP
    port: 80
EOF
```

2. Observe your service. Note that it does not have a cluster IP.

```
$ kubectl get svc my-httpbin
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT
(S) AGE				
my-httpbin	ExternalName	<none>	httpbin.org	80/TCP
CP	4s			

3. Access `httpbin.org` via the Kubernetes service's hostname from the source pod without Istio sidecar. Note that the `curl` command below uses the Kubernetes DNS format for services: `<service name>. <namespace>.svc.cluster.local.`

```
$ kubectl exec "$SOURCE_POD_WITHOUT_ISTIO" -n without-istio
  -c sleep -- curl -sS my-httpbin.org.default.svc.cluster.local/
headers
{
  "headers": {
    "Accept": "*/*",
    "Host": "my-httpbin.org.default.svc.cluster.local",
    "User-Agent": "curl/7.55.0"
  }
}
```

4. In this example, unencrypted HTTP requests are sent to [httpbin.org](http://httpbin.org). For the sake of the example only, you disable the TLS mode and allow the unencrypted traffic to the external service. In the real life scenarios, we recommend to perform Egress TLS origination by Istio.

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: my-httpbin
spec:
  host: my-httpbin.default.svc.cluster.local
  trafficPolicy:
    tls:
      mode: DISABLE
EOF
```

5. Access [httpbin.org](http://httpbin.org) via the Kubernetes service's hostname from the source pod with Istio sidecar. Notice the headers added by Istio sidecar, for example `X-Envoy-Decorator-Operation`. Also note that the `Host` header equals to your service's

hostname.

```
$ kubectl exec "$SOURCE_POD" -c sleep -- curl -sS my-httppbin.default.svc.cluster.local/headers
{
  "headers": {
    "Accept": "*/*",
    "Content-Length": "0",
    "Host": "my-httppbin.default.svc.cluster.local",
    "User-Agent": "curl/7.64.0",
    "X-B3-Sampled": "0",
    "X-B3-Spanid": "5795fab599dca0b8",
    "X-B3-Traceid": "5079ad3a4af418915795fab599dca0b8",
    "X-Envoy-Decorator-Operation": "my-httppbin.default.svc.cluster.local:80/*",
    "X-Envoy-Peer-Metadata": "...",
    "X-Envoy-Peer-Metadata-Id": "sidecar~10.28.1.74~sleep-6bdb595bcb-drr45.default~default.svc.cluster.local"
  }
}
```

# **Cleanup of Kubernetes ExternalName service**

```
$ kubectl delete destinationrule my-httpbin  
$ kubectl delete service my-httpbin
```

**Use a Kubernetes service  
with endpoints to access an  
external service**

1. Create a Kubernetes service without selector for Wikipedia:

```
$ kubectl apply -f - <<EOF
kind: Service
apiVersion: v1
metadata:
  name: my-wikipedia
spec:
  ports:
    - protocol: TCP
      port: 443
      name: tls
EOF
```

2. Create endpoints for your service. Pick a couple of IPs from the Wikipedia ranges list.

```
$ kubectl apply -f - <<EOF
kind: Endpoints
apiVersion: v1
metadata:
  name: my-wikipedia
subsets:
- addresses:
  - ip: 91.198.174.192
  - ip: 198.35.26.96
  ports:
  - port: 443
    name: tls
EOF
```

3. Observe your service. Note that it has a cluster IP which you can use to access wikipedia.org.

```
$ kubectl get svc my-wikipedia
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	P
PORT(S)	AGE			
my-wikipedia	ClusterIP	172.21.156.230	<none>	4
43/TCP	21h			

4. Send HTTPS requests to `wikipedia.org` by your Kubernetes service's cluster IP from the source pod without Istio sidecar. Use the `--resolve` option of `curl` to access `wikipedia.org` by the cluster IP:

```
$ kubectl exec "$SOURCE_POD_WITHOUT_ISTIO" -n without-istio  
-c sleep -- curl -sS --resolve en.wikipedia.org:443:"$(kub  
ectl get service my-wikipedia -o jsonpath='{.spec.clusterIP  
'})" https://en.wikipedia.org/wiki/Main_Page | grep -o "<ti  
tle>.*</title>"  
<title>Wikipedia, the free encyclopedia</title>
```

5. In this case, the workload send HTTPS requests (open TLS connection) to the wikipedia.org. The traffic is already encrypted by the workload so you can safely disable Istio's mutual TLS:

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: my-wikipedia
spec:
  host: my-wikipedia.default.svc.cluster.local
  trafficPolicy:
    tls:
      mode: DISABLE
EOF
```

6. Access `wikipedia.org` by your Kubernetes service's cluster IP from the source pod with Istio sidecar:

```
$ kubectl exec "$SOURCE_POD" -c sleep -- curl -sS --resolve en.wikipedia.org:443:"$(kubectl get service my-wikipedia -o jsonpath='{.spec.clusterIP}')" https://en.wikipedia.org/w/index/Main_Page | grep -o "<title>.*</title>"<title>Wikipedia, the free encyclopedia</title>
```

7. Check that the access is indeed performed by the cluster IP. Notice the sentence `Connected to en.wikipedia.org (172.21.156.230)` in the output of `curl -v`, it mentions the IP that was printed in the output of your service as the cluster IP.

```
$ kubectl exec "$SOURCE_POD" -c sleep -- curl -sS -v --resolv en.wikipedia.org:443:"$(kubectl get service my-wikipedia -o jsonpath='{.spec.clusterIP}')" https://en.wikipedia.org/wiki/Main_Page -o /dev/null
* Added en.wikipedia.org:443:172.21.156.230 to DNS cache
* Hostname en.wikipedia.org was found in DNS cache
* Trying 172.21.156.230...
* TCP_NODELAY set
* Connected to en.wikipedia.org (172.21.156.230) port 443 (#0)
...
...
```

# Cleanup of Kubernetes service with endpoints

```
$ kubectl delete destinationrule my-wikipedia  
$ kubectl delete endpoints my-wikipedia  
$ kubectl delete service my-wikipedia
```

# Cleanup

1. Shutdown the sleep service:

```
$ kubectl delete -f @samples/sleep/sleep.yaml@
```

2. Shutdown the sleep service in the without-istio namespace:

```
$ kubectl delete -f @samples/sleep/sleep.yaml@ -n without-istio
```

### 3. Delete without-istio namespace:

```
$ kubectl delete namespace without-istio
```

### 4. Unset the environment variables:

```
$ unset SOURCE_POD SOURCE_POD_WITHOUT_ISTIO
```

# Using an External HTTPS Proxy

⌚ 5 minute read ✓ page test

---

The [Configure an Egress Gateway](#) example shows how to direct traffic to external services from your mesh via an Istio edge component called *Egress Gateway*. However, some cases require an external, legacy (non-Istio) HTTPS proxy to access external services.

For example, your company may already have such a proxy in place and all the applications within the organization may be required to direct their traffic through it.

This example shows how to enable access to an external HTTPS proxy. Since applications use the HTTP CONNECT method to establish connections with HTTPS proxies, configuring traffic to an external HTTPS proxy is different from configuring traffic to external HTTP and HTTPS services.

# Before you begin

- Setup Istio by following the instructions in the Installation guide.



The egress gateway and access logging will be enabled if you install the demo configuration profile.

- Deploy the sleep sample app to use as a test source for sending requests. If you have automatic

sidecar injection enabled, run the following command to deploy the sample app:

```
$ kubectl apply -f @samples/sleep/sleep.yaml@
```

Otherwise, manually inject the sidecar before deploying the sleep application with the following command:

```
$ kubectl apply -f <(istioctl kube-inject -f @samples/sleep/sleep.yaml@)
```



You can use any pod with curl installed as a test source.

- Set the `SOURCE_POD` environment variable to the name of your source pod:

```
$ export SOURCE_POD=$(kubectl get pod -l app=sleep -o jsonpath={.items..metadata.name})
```

- Enable Envoy's access logging

## Deploy an HTTPS proxy

To simulate a legacy proxy and only for this example,

you deploy an HTTPS proxy inside your cluster. Also, to simulate a more realistic proxy that is running outside of your cluster, you will address the proxy's pod by its IP address and not by the domain name of a Kubernetes service. This example uses Squid but you can use any HTTPS proxy that supports HTTP CONNECT.

1. Create a namespace for the HTTPS proxy, without labeling it for sidecar injection. Without the label, sidecar injection is disabled in the new namespace so Istio will not control the traffic there. You need this behavior to simulate the proxy being outside of the cluster.

```
$ kubectl create namespace external
```

## 2. Create a configuration file for the Squid proxy.

```
$ cat <<EOF > ./proxy.conf
http_port 3128

acl SSL_ports port 443
acl CONNECT method CONNECT

http_access deny CONNECT !SSL_ports
http_access allow localhost manager
http_access deny manager
http_access allow all

coredump_dir /var/spool/squid
EOF
```

### 3. Create a Kubernetes ConfigMap to hold the configuration of the proxy:

```
$ kubectl create configmap proxy-configmap -n external --from-file=squid.conf=./proxy.conf
```

### 4. Deploy a container with Squid:

```
$ kubectl apply -f - <<EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: squid
  namespace: external
spec:
  replicas: 1
  selector:
    matchLabels:
```

```
    app: squid
template:
  metadata:
    labels:
      app: squid
spec:
  volumes:
    - name: proxy-config
      configMap:
        name: proxy-configmap
  containers:
    - name: squid
      image: sameersbn/squid:3.5.27
      imagePullPolicy: IfNotPresent
      volumeMounts:
        - name: proxy-config
          mountPath: /etc/squid
          readOnly: true
```

EOF

5. Deploy the sleep sample in the external namespace to test traffic to the proxy without Istio traffic control.

```
$ kubectl apply -n external -f @samples/sleep/sleep.yaml@
```

6. Obtain the IP address of the proxy pod and define the PROXY\_IP environment variable to store it:

```
$ export PROXY_IP="$(kubectl get pod -n external -l app=squid -o jsonpath={.items..podIP})"
```

7. Define the PROXY\_PORT environment variable to store the port of your proxy. In this case, Squid uses port 3128.

```
$ export PROXY_PORT=3128
```

8. Send a request from the sleep pod in the external namespace to an external service via the proxy:

```
$ kubectl exec "$(kubectl get pod -n external -l app=sleep -o jsonpath={.items..metadata.name})" -n external -- sh -c "HTTPS_PROXY=$PROXY_IP:$PROXY_PORT curl https://en.wikipedia.org/wiki/Main_Page" | grep -o "<title>.*</title>"<title>Wikipedia, the free encyclopedia</title>
```

9. Check the access log of the proxy for your request:

```
$ kubectl exec "$(kubectl get pod -n external -l app=squid -o jsonpath={.items..metadata.name})" -n external -- tail /var/log/squid/access.log
1544160065.248      228 172.30.109.89 TCP_TUNNEL/200 87633 CONNECT en.wikipedia.org:443 - HIER_DIRECT/91.198.174.192 -
```

So far, you completed the following tasks without Istio:

- You deployed the HTTPS proxy.
- You used curl to access the wikipedia.org external service through the proxy.

Next, you must configure the traffic from the Istio-enabled pods to use the HTTPS proxy.

# Configure traffic to external HTTPS proxy

1. Define a TCP (not HTTP!) Service Entry for the HTTPS proxy. Although applications use the HTTP CONNECT method to establish connections with HTTPS proxies, you must configure the proxy for TCP traffic, instead of HTTP. Once the connection is established, the proxy simply acts as a TCP tunnel.

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1beta1
kind: ServiceEntry
metadata:
  name: proxy
spec:
  hosts:
    - my-company-proxy.com # ignored
  addresses:
    - $PROXY_IP/32
  ports:
    - number: $PROXY_PORT
      name: tcp
      protocol: TCP
  location: MESH_EXTERNAL
EOF
```

2. Send a request from the sleep pod in the default namespace. Because the sleep pod has a sidecar,

# Istio controls its traffic.

```
$ kubectl exec "$SOURCE_POD" -c sleep -- sh -c "HTTPS_PROXY=$PROXY_IP:$PROXY_PORT curl https://en.wikipedia.org/wiki/Main_Page" | grep -o "<title>.*</title>"  
<title>Wikipedia, the free encyclopedia</title>
```

3. Check the Istio sidecar proxy's logs for your request:

```
$ kubectl logs "$SOURCE_POD" -c istio-proxy  
[2018-12-07T10:38:02.841Z] " - - - " 0 - 702 87599 92 - "-" "  
- " "-" "-" "172.30.109.95:3128" outbound|3128||my-company-p  
roxy.com 172.30.230.52:44478 172.30.109.95:3128 172.30.230.  
52:44476 -
```

4. Check the access log of the proxy for your

request:

```
$ kubectl exec "$(kubectl get pod -n external -l app=squid  
-o jsonpath={.items..metadata.name})" -n external -- tail /  
var/log/squid/access.log  
1544160065.248      228 172.30.109.89 TCP_TUNNEL/200 87633 CO  
NNECT en.wikipedia.org:443 - HIER_DIRECT/91.198.174.192 -
```

## Understanding what happened

In this example, you took the following steps:

1. Deployed an HTTPS proxy to simulate an external proxy.
2. Created a TCP service entry to enable Istio-controlled traffic to the external proxy.

Note that you must not create service entries for the external services you access through the external proxy, like `wikipedia.org`. This is because from Istio's point of view the requests are sent to the external proxy only; Istio is not aware of the fact that the external proxy forwards the requests further.

# Cleanup

1. Shutdown the sleep service:

```
$ kubectl delete -f @samples/sleep/sleep.yaml@
```

2. Shutdown the sleep service in the external namespace:

```
$ kubectl delete -f @samples/sleep/sleep.yaml@ -n external
```

3. Shutdown the Squid proxy, remove the ConfigMap and the configuration file:

```
$ kubectl delete -n external deployment squid  
$ kubectl delete -n external configmap proxy-configmap  
$ rm ./proxy.conf
```

#### 4. Delete the external namespace:

```
$ kubectl delete namespace external
```

#### 5. Delete the Service Entry:

```
$ kubectl delete serviceentry proxy
```

# Plug in CA Certificates

⌚ 4 minute read ✓ page test

---

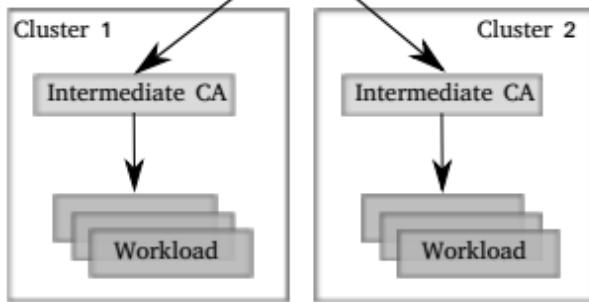
This task shows how administrators can configure the Istio certificate authority (CA) with a root certificate, signing certificate and key.

By default the Istio CA generates a self-signed root certificate and key and uses them to sign the

workload certificates. To protect the root CA key, you should use a root CA which runs on a secure machine offline, and use the root CA to issue intermediate certificates to the Istio CAs that run in each cluster. An Istio CA can sign workload certificates using the administrator-specified certificate and key, and distribute an administrator-specified root certificate to the workloads as the root of trust.

The following graph demonstrates the recommended CA hierarchy in a mesh containing two clusters.





## CA Hierarchy

This task demonstrates how to generate and plug in the certificates and key for the Istio CA. These steps can be repeated to provision certificates and keys for Istio CAs running in each cluster.

# Plug in certificates and key into the cluster



The following instructions are for demo purposes only. For a production cluster setup, it is highly recommended to use a production-ready CA, such as Hashicorp Vault. It is a good practice to manage the root CA on an offline machine with strong security protection.

1. In the top-level directory of the Istio installation package, create a directory to hold certificates and keys:

```
$ mkdir -p certs  
$ pushd certs
```

2. Generate the root certificate and key:

```
$ make -f ../../tools/certs/Makefile.selfsigned.mk root-ca
```

This will generate the following files:

- `root-cert.pem`: the generated root certificate
- `root-key.pem`: the generated root key
- `root-ca.conf`: the configuration for `openssl` to

generate the root certificate

- `root-cert.csr`: the generated CSR for the root certificate
3. For each cluster, generate an intermediate certificate and key for the Istio CA. The following is an example for `cluster1`:

```
$ make -f ../tools/certs/Makefile.signed.mk cluster1-ca certs
```

This will generate the following files in a directory named `cluster1`:

- `ca-cert.pem`: the generated intermediate certificates

- `ca-key.pem`: the generated intermediate key
- `cert-chain.pem`: the generated certificate chain which is used by istiod
- `root-cert.pem`: the root certificate

You can replace `cluster1` with a string of your choosing. For example, with the argument `cluster2-cacerts`, you can create certificates and key in a directory called `cluster2`.

If you are doing this on an offline machine, copy the generated directory to a machine with access to the clusters.

4. In each cluster, create a secret `cacerts` including

all the input files ca-cert.pem, ca-key.pem, root-cert.pem and cert-chain.pem. For example, for cluster1:

```
$ kubectl create namespace istio-system  
$ kubectl create secret generic cacerts -n istio-system \  
    --from-file=cluster1/ca-cert.pem \  
    --from-file=cluster1/ca-key.pem \  
    --from-file=cluster1/root-cert.pem \  
    --from-file=cluster1/cert-chain.pem
```

5. Return to the top-level directory of the Istio installation:

```
$ popd
```

# Deploy Istio

1. Deploy Istio using the `demo` profile.

Istio's CA will read certificates and key from the secret-mount files.

```
$ istioctl install --set profile=demo
```

## Deploying example services

## 1. Deploy the httpbin and sleep sample services.

```
$ kubectl create ns foo
$ kubectl apply -f <(istioctl kube-inject -f samples/httpbin/httpbin.yaml) -n foo
$ kubectl apply -f <(istioctl kube-inject -f samples/sleep/sleep.yaml) -n foo
```

## 2. Deploy a policy for workloads in the foo namespace to only accept mutual TLS traffic.

```
$ kubectl apply -n foo -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: "default"
spec:
  mtls:
    mode: STRICT
EOF
```

# Verifying the certificates

In this section, we verify that workload certificates are signed by the certificates that we plugged into the

CA. This requires you have `openssl` installed on your machine.

1. Sleep 20 seconds for the mTLS policy to take effect before retrieving the certificate chain of `httpbin`. As the CA certificate used in this example is self-signed, the `verify error:num=19:self signed certificate in certificate chain` error returned by the `openssl` command is expected.

```
$ sleep 20; kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})" -c istio-proxy -n foo -- openssl s_client -showcerts -connect httpbin.foo:8000 > httpbin-proxy-cert.txt
```

## 2. Parse the certificates on the certificate chain.

```
$ sed -n '/-----BEGIN CERTIFICATE-----/{:start /-----END CE  
RTIFICATE-----/!{N;b start};/.*/p}' httpbin-proxy-cert.txt  
> certs.pem  
$ awk 'BEGIN {counter=0;} /BEGIN CERT/{counter++} { print >  
"proxy-cert-" counter ".pem"}' < certs.pem
```

## 3. Verify the root certificate is the same as the one specified by the administrator:

```
$ openssl x509 -in certs/cluster1/root-cert.pem -text -noou  
t > /tmp/root-cert.crt.txt  
$ openssl x509 -in ./proxy-cert-3.pem -text -noout > /tmp/p  
od-root-cert.crt.txt  
$ diff -s /tmp/root-cert.crt.txt /tmp/pod-root-cert.crt.txt  
Files /tmp/root-cert.crt.txt and /tmp/pod-root-cert.crt.txt  
are identical
```

4. Verify the CA certificate is the same as the one specified by the administrator:

```
$ openssl x509 -in certs/cluster1/ca-cert.pem -text -noout > /tmp/ca-cert.crt.txt  
$ openssl x509 -in ./proxy-cert-2.pem -text -noout > /tmp/pod-cert-chain-ca.crt.txt  
$ diff -s /tmp/ca-cert.crt.txt /tmp/pod-cert-chain-ca.crt.txt  
Files /tmp/ca-cert.crt.txt and /tmp/pod-cert-chain-ca.crt.txt are identical
```

5. Verify the certificate chain from the root certificate to the workload certificate:

```
$ openssl verify -CAfile <(cat certs/cluster1/ca-cert.pem certs/cluster1/root-cert.pem) ./proxy-cert-1.pem  
./proxy-cert-1.pem: OK
```

# Cleanup

- Remove the certificates, keys, and intermediate files from your local disk:

```
$ rm -rf certs
```

- Remove the secret cacerts, and the foo and istio-

## system namespaces:

```
$ kubectl delete secret cacerts -n istio-system  
$ kubectl delete ns foo istio-system
```

- To remove the Istio components: follow the uninstall instructions **to remove**.

# Istio DNS Certificate Management

⌚ 3 minute read ✓ page test

---

This task shows how to provision and manage DNS certificates using Chiron, a lightweight component linked with Istiod that signs certificates using the Kubernetes CA APIs without maintaining its own private key. Using this feature has the following

advantages:

- Unlike Istiod, this feature doesn't require maintaining a private signing key, which enhances security.
- Simplified root certificate distribution to TLS clients. Clients no longer need to wait for Istiod to generate and distribute its CA certificate.

**Before you begin**

- Install Istio through `istioctl` with DNS certificates configured. The configuration is read when Istiod starts.

```
$ cat <<EOF > ./istio.yaml
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  meshConfig:
    certificates:
      - secretName: dns.example1-service-account
        dnsNames: [example1.istio-system.svc, example1.istio-sys
tem]
      - secretName: dns.example2-service-account
        dnsNames: [example2.istio-system.svc, example2.istio-sys
tem]
EOF
$ istioctl install -f ./istio.yaml
```

# DNS certificate provisioning and management

Istio provisions the DNS names and secret names for the DNS certificates based on configuration you provide. The DNS certificates provisioned are signed by the Kubernetes CA and stored in the secrets following your configuration. Istio also manages the lifecycle of the DNS certificates, including their rotations and regenerations.

# Configure DNS certificates

The `IstioOperator` custom resource used to configure Istio in the `istioctl install` command, above, contains an example DNS certificate configuration. Within, the `dnsNames` field specifies the DNS names in a certificate and the `secretName` field specifies the name of the Kubernetes secret used to store the certificate and the key.

## Check the provisioning of

# DNS certificates

After configuring Istio to generate DNS certificates and storing them in secrets of your choosing, you can verify that the certificates were provisioned and work properly.

To check that Istio generated the `dns.example1-service-account` DNS certificate as configured in the example, and that the certificate contains the configured DNS names, you need to get the secret from Kubernetes, parse it, decode it, and view its text output with the following command:

```
$ kubectl get secret dns.example1-service-account -n istio-system -o jsonpath='{.data['cert-chain\.pem']}'}' | base64 --decode | openssl x509 -in /dev/stdin -text -noout
```

The text output should include:

```
X509v3 Subject Alternative Name:  
DNS:example1.istio-system.svc, DNS:example1.istio-system
```

# Regenerating a DNS certificate

Istio can also regenerate DNS certificates that were mistakenly deleted. Next, we show how you can delete a recently configured certificate and verify Istio regenerates it automatically.

1. Delete the secret storing the DNS certificate configured earlier:

```
$ kubectl delete secret dns.example1-service-account -n istio-system
```

2. To check that Istio regenerated the deleted DNS certificate, and that the certificate contains the configured DNS names, you need to get the secret from Kubernetes, parse it, decode it, and

view its text output with the following command:

```
$ sleep 10; kubectl get secret dns.example1-service-account  
-n istio-system -o jsonpath=".data['cert-chain\\.pem']" |  
base64 --decode | openssl x509 -in /dev/stdin -text -noout
```

The output should include:

X509v3 Subject Alternative Name:

DNS:example1.istio-system.svc, DNS:example1.istio-system

# Cleanup

- To remove the `istio-system` namespace:

```
$ kubectl delete ns istio-system
```

# Custom CA Integration using Kubernetes CSR

⌚ 6 minute read  page test

---

---



This feature is actively in development and is considered experimental.

This feature requires Kubernetes version  $\geq 1.18$ .

This task shows how to provision Workload Certificates using a custom certificate authority that integrates with the Kubernetes CSR API. This feature leverages Chiron, a lightweight component linked with Istiod that signs certificates using the Kubernetes CSR API.

This task is split into two parts. The first part demonstrates how to use the Kubernetes CA itself to sign workload certificates. The second part demonstrates how to use a custom CA that integrates

with the Kubernetes CSR API to sign your certificates.

## Part 1: Using Kubernetes CA



Note that this example should only be used for basic evaluation. The use of the [kubernetes.io/legacy-unknown](https://kubernetes.io/legacy-unknown) signer is NOT recommended in production environments.

# **Deploying Istio with Kubernetes CA**

1. Deploy Istio on the cluster using `istioctl` with the following configuration.

```
$ cat <<EOF > ./istio.yaml
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  components:
    pilot:
      k8s:
        env:
          # Indicate to Istiod that we use a Custom Certificate Authority
          - name: EXTERNAL_CA
            value: ISTIOD_RA_KUBERNETES_API
          # Tells Istiod to use the Kubernetes legacy CA Signer
          - name: K8S_SIGNER
            value: kubernetes.io/legacy-unknown
EOF
$ istioctl install --set profile=demo -f ./istio.yaml
```

2. Deploy the bookinfo sample application in the bookinfo namespace. Ensure that the following commands are executed in the Istio root directory.

```
$ kubectl create ns bookinfo
$ kubectl apply -f <(istioctl kube-inject -f samples/bookinfo/platform/kube/bookinfo.yaml) -n bookinfo
```

**Verify that the certificates installed are correct**

When the workloads are deployed, above, they send CSR Requests to Istiod which forwards them to the Kubernetes CA for signing. If all goes well, the signed certificates are sent back to the workloads where they are then installed. To verify that they have been signed by the Kubernetes CA, you need to first extract the signed certificates.

1. Dump all pods running in the namespace.

```
$ kubectl get pods -n bookinfo
```

Pick any one of the running pods for the next step.

2. Get the certificate chain and CA root certificate used by the Istio proxies for mTLS.

```
$ istioctl pc secret <pod-name> -o json > proxy_secret
```

The proxy\_secret json file contains the CA root certificate for mTLS in the trustedCA field. Note that this certificate is base64 encoded.

3. The certificate used by the Kubernetes CA (specifically the kubernetes.io/legacy-unknown signer) is loaded onto the secret associated with every service account in the bookinfo namespace.

```
$ kubectl get secrets -n bookinfo
```

Pick a secret-name that is associated with any of the service-accounts. These have a “token” in their name.

```
$ kubectl get secrets -n bookinfo <secret-name> -o json
```

The `ca.crt` field in the output contains the base64 encoded Kubernetes CA certificate.

4. Compare the `ca.cert` obtained in the previous step with the contents of the `TrustedCA` field in the step before. These two should be the same.
5. (Optional) Follow the rest of the steps in the bookinfo example to ensure that communication between services is working as expected.

# Cleanup Part 1

- Remove the `istio-system` and `bookinfo` namespaces:

```
$ kubectl delete ns istio-system  
$ kubectl delete ns bookinfo
```

## Part 2: Using Custom CA

This assumes that the custom CA implements a controller that has the necessary permissions to read

and sign Kubernetes CSR Requests. Refer to the Kubernetes CSR documentation for more details. Note that the steps below are dependent on an external-source and may change.

## **Deploy Custom CA controller in the Kubernetes cluster**

1. For this example, we use an open-source Certificate Authority implementation. This code builds a

controller that reads the CSR resources on the Kubernetes cluster and creates certificates using local keys. Follow the instructions on the page to:

1. Build the Certificate-Controller docker image
  2. Upload the image to a Docker Registry
  3. Generate the Kubernetes manifest to deploy it
- 
2. Deploy the Kubernetes manifest generated in the previous step on your local cluster in the signer-ca-system namespace.

```
$ kubectl apply -f local-ca.yaml
```

Ensure that all the services are running.

```
$ kubectl get services -n signer-ca-system
  NAME                                     TYPE
  CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
  signer-ca-controller-manager-metrics-service   ClusterIP
  10.8.9.25      none         8443/TCP   72s
```

3. Get the public key of the CA. This is encoded in the secret “signer-ca-\*” in the signer-ca-system namespace.

```
$ kubectl get secrets signer-ca-5hff5h74hm -n signer-ca-system -o json
```

The `tls.crt` field contains the base64 encoded public key file. Record this for future use.

# **Load the CA root certificate into a secret that istiod can access**

1. Load the secret into the istiod namespace.

```
$ cat <<EOF > ./external-ca-secret.yaml
apiVersion: v1
kind: Secret
metadata:
  name: external-ca-cert
  namespace: istio-system
data:
  root-cert.pem: <tls.cert from the step above>
EOF
$ kubectl apply -f external-ca-secret.yaml
```

This step is necessary for Istio to verify that the workload certificates have been signed by the correct certificate authority and to add the root-cert to the trust bundle for mTLS to work.

# Deploying Istio

1. Deploy Istio on the cluster using `istioctl` with the following configuration.

```
$ cat <<EOF > ./istio.yaml
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  components:
    pilot:
      k8s:
        env:
          # Indicate to Istiod that we use an external sign
er
          - name: EXTERNAL_CA
            value: ISTIOD_RA_KUBERNETES_API
```

```
# Indicate to Istiod the external k8s Signer Name
- name: K8S_SIGNER
  value: example.com/foo
overlays:
  # Amend ClusterRole to add permission for istiod
  to approve certificate signing by custom signer
  - kind: ClusterRole
    name: istiod-clusterrole-istio-system
  patches:
    - path: rules[-1]
      value: |
        apiGroups:
          - certificates.k8s.io
        resourceNames:
          - example.com/foo
        resources:
          - signers
        verbs:
          - approve
- kind: Deployment
```

```
        name: istiod
        patches:
          - path: spec.template.spec.containers[0].volumeMounts[-1]
            value: |
              # Mount external CA certificate into Istiod
              name: external-ca-cert
              mountPath: /etc/external-ca-cert
              readOnly: true
          - path: spec.template.spec.volumes[-1]
            value: |
              name: external-ca-cert
              secret:
                secretName: external-ca-cert
                optional: true
EOF
$ istioctl install --set profile=demo -f ./istio.yaml
```

## 2. Deploy the bookinfo sample application in the

bookinfo namespace.

```
$ kubectl create ns bookinfo
$ kubectl apply -f <(istioctl kube-inject -f samples/bookinfo/platform/kube/bookinfo.yaml) -n bookinfo
```

## Verify that Custom CA certificates installed are correct

When the workloads are deployed, above, they send CSR Requests to Istiod which forwards them to the

Kubernetes CA for signing. If all goes well, the signed certificates are sent back to the workloads where they are then installed. To verify that they have indeed been signed by the Kubernetes CA, you need to first extract the signed certificates.

1. Dump all pods running in the namespace.

```
$ kubectl get pods -n bookinfo
```

Pick any of the running pods for the next step.

2. Get the certificate chain and CA root certificate used by the Istio proxies for mTLS.

```
$ istioctl pc secret <pod-name> -n bookinfo -o json > proxy  
_secret
```

The proxy\_secret json file contains the CA root certificate for mTLS in the trustedCA field. Note that this certificate is base64 encoded.

3. Compare the CA root certificate obtained in the step above with “root-cert.pem” value in external-ca-cert. These two should be the same.
4. (Optional) Follow the rest of the steps in the bookinfo example to ensure that communication between services is working as expected.

# Cleanup Part 2

- Remove the `istio-system` and `bookinfo` namespaces:

```
$ kubectl delete ns istio-system  
$ kubectl delete ns bookinfo
```

## Reasons to use this feature

- Added Security - Unlike `plugin-ca-cert` or the default `self-signed` option, enabling this feature

means that the CA private keys need not be present in the Kubernetes cluster.

- Custom CA Integration - By specifying a Signer name in the Kubernetes CSR Request, this feature allows Istio to integrate with custom Certificate Authorities using the Kubernetes CSR API interface. This does require the custom CA to implement a Kubernetes controller to watch the `CertificateSigningRequest` and `Certificate` Resources and act on them.



# Authentication Policy

⌚ 13 minute read ✓ page test

---

This task covers the primary activities you might need to perform when enabling, configuring, and using Istio authentication policies. Find out more about the underlying concepts in the authentication overview.

# Before you begin

- Understand Istio authentication policy and related mutual TLS authentication concepts.
- Install Istio on a Kubernetes cluster with the default configuration profile, as described in installation steps.

```
$ istioctl install --set profile=default
```

# Setup

Our examples use two namespaces `foo` and `bar`, with two services, `httpbin` and `sleep`, both running with an Envoy proxy. We also use second instances of `httpbin` and `sleep` running without the sidecar in the legacy namespace. If you'd like to use the same examples when trying the tasks, run the following:

```
$ kubectl create ns foo
$ kubectl apply -f <(istioctl kube-inject -f @samples/httpbin/httpbin.yaml@) -n foo
$ kubectl apply -f <(istioctl kube-inject -f @samples/sleep/sleep.yaml@) -n foo
$ kubectl create ns bar
$ kubectl apply -f <(istioctl kube-inject -f @samples/httpbin/httpbin.yaml@) -n bar
$ kubectl apply -f <(istioctl kube-inject -f @samples/sleep/sleep.yaml@) -n bar
$ kubectl create ns legacy
$ kubectl apply -f @samples/httpbin/httpbin.yaml@ -n legacy
$ kubectl apply -f @samples/sleep/sleep.yaml@ -n legacy
```

You can verify setup by sending an HTTP request with curl from any sleep pod in the namespace foo, bar or legacy to either httpbin.foo, httpbin.bar or

httpbin.legacy. All requests should succeed with HTTP code 200.

For example, here is a command to check sleep.bar to httpbin.foo reachability:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n bar -o jsonpath={.items..metadata.name})" -c sleep -n bar -- curl http://httpbin.foo:8000/ip -s -o /dev/null -w "%{http_code}\n"  
200
```

This one-liner command conveniently iterates through all reachability combinations:

```
$ for from in "foo" "bar" "legacy"; do for to in "foo" "bar" "legacy"; do kubectl exec "$(kubectl get pod -l app=sleep -n ${from} -o jsonpath={.items..metadata.name})" -c sleep -n ${from} -- curl -s "http://httpbin.${to}:8000/ip" -s -o /dev/null -w "sleep.${from} to httpbin.${to}: %{http_code}\n"; done; done  
sleep.foo to httpbin.foo: 200  
sleep.foo to httpbin.bar: 200  
sleep.foo to httpbin.legacy: 200  
sleep.bar to httpbin.foo: 200  
sleep.bar to httpbin.bar: 200  
sleep.bar to httpbin.legacy: 200  
sleep.legacy to httpbin.foo: 200  
sleep.legacy to httpbin.bar: 200  
sleep.legacy to httpbin.legacy: 200
```

Verify there is no peer authentication policy in the system with the following command:

```
$ kubectl get peerauthentication --all-namespaces  
No resources found
```

Last but not least, verify that there are no destination rules that apply on the example services. You can do this by checking the `host:` value of existing destination rules and make sure they do not match. For example:

```
$ kubectl get destinationrules.networking.istio.io --all-namespaces -o yaml | grep "host:"
```

Depending on the version of Istio, you may see destination rules for hosts other than

- i) those shown. However, there should be none with hosts in the foo, bar and legacy namespace, nor is the match-all wildcard \*

## Auto mutual TLS

By default, Istio tracks the server workloads migrated to Istio proxies, and configures client proxies to send mutual TLS traffic to those workloads automatically, and to send plain text traffic to workloads without

sidecars.

Thus, all traffic between workloads with proxies uses mutual TLS, without you doing anything. For example, take the response from a request to `httpbin/header`. When using mutual TLS, the proxy injects the `X-Forwarded-Client-Cert` header to the upstream request to the backend. That header's presence is evidence that mutual TLS is used. For example:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})" -c sleep -n foo -- curl -s http://httpbin.foo:8000/headers -s | grep X-Forwarded-Client-Cert | sed 's/Hash=[a-zA-Z0-9]*;/Hash=<redacted>;/'  
"X-Forwarded-Client-Cert": "By=spiffe://cluster.local/ns/foo/sa/httpbin;Hash=<redacted>;Subject=\"\";URI=spiffe://cluster.local/ns/foo/sa/sleep"
```

When the server doesn't have sidecar, the X-Forwarded-Client-Cert header is not there, which implies requests are in plain text.

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})" -c sleep -n foo -- curl http://httpbin.legacy:8000/headers -s | grep X-Forwarded-Client-Cert
```

# Globally enabling Istio mutual TLS in STRICT mode

While Istio automatically upgrades all traffic between the proxies and the workloads to mutual TLS, workloads can still receive plain text traffic. To prevent non-mutual TLS traffic for the whole mesh, set a mesh-wide peer authentication policy with the mutual TLS mode set to `STRICT`. The mesh-wide peer authentication policy should not have a `selector` and must be applied in the **root namespace**, for example:

```
$ kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: "default"
  namespace: "istio-system"
spec:
  mtls:
    mode: STRICT
EOF
```

 The example assumes `istio-system` is the root namespace. If you used a different value during installation, replace `istio-system` with the value you used.

This peer authentication policy configures workloads to only accept requests encrypted with TLS. Since it doesn't specify a value for the `selector` field, the policy applies to all workloads in the mesh.

Run the test command again:

```
$ for from in "foo" "bar" "legacy"; do for to in "foo" "bar" "legacy"; do kubectl exec "$(kubectl get pod -l app=sleep -n ${from} -o jsonpath={.items..metadata.name})" -c sleep -n ${from} -- curl "http://httpbin.${to}:8000/ip" -s -o /dev/null -w "sleep.${from} to httpbin.${to}: %{http_code}\n"; done; done
sleep.foo to httpbin.foo: 200
sleep.foo to httpbin.bar: 200
sleep.foo to httpbin.legacy: 200
sleep.bar to httpbin.foo: 200
sleep.bar to httpbin.bar: 200
sleep.bar to httpbin.legacy: 200
sleep.legacy to httpbin.foo: 000
command terminated with exit code 56
sleep.legacy to httpbin.bar: 000
command terminated with exit code 56
sleep.legacy to httpbin.legacy: 200
```

You see requests still succeed, except for those from

the client that doesn't have proxy, sleep.legacy, to the server with a proxy, httpbin.foo or httpbin.bar. This is expected because mutual TLS is now strictly required, but the workload without sidecar cannot comply.

## Cleanup part 1

Remove global authentication policy and destination rules added in the session:

```
$ kubectl delete peerauthentication -n istio-system default
```

# Enable mutual TLS per namespace or workload

## Namespace-wide policy

To change mutual TLS for all workloads within a particular namespace, use a namespace-wide policy. The specification of the policy is the same as for a mesh-wide policy, but you specify the namespace it applies to under `metadata`. For example, the following peer authentication policy enables strict mutual TLS

for the `foo` namespace:

```
$ kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: "default"
  namespace: "foo"
spec:
  mtls:
    mode: STRICT
EOF
```

As this policy is applied on workloads in namespace `foo` only, you should see only request from client-`without-sidecar` (`sleep.legacy`) to `httpbin.foo` start to fail.

```
$ for from in "foo" "bar" "legacy"; do for to in "foo" "bar" "legacy"; do kubectl exec "$(kubectl get pod -l app=sleep -n ${from} -o jsonpath={.items..metadata.name})" -c sleep -n ${from} -- curl "http://httpbin.${to}:8000/ip" -s -o /dev/null -w "sleep.${from} to httpbin.${to}: %{http_code}\n"; done; done
sleep.foo to httpbin.foo: 200
sleep.foo to httpbin.bar: 200
sleep.foo to httpbin.legacy: 200
sleep.bar to httpbin.foo: 200
sleep.bar to httpbin.bar: 200
sleep.bar to httpbin.legacy: 200
sleep.legacy to httpbin.foo: 000
command terminated with exit code 56
sleep.legacy to httpbin.bar: 200
sleep.legacy to httpbin.legacy: 200
```

# Enable mutual TLS per

# workload

To set a peer authentication policy for a specific workload, you must configure the `selector` section and specify the labels that match the desired workload. However, Istio cannot aggregate workload-level policies for outbound mutual TLS traffic to a service. Configure a destination rule to manage that behavior.

For example, the following peer authentication policy and destination rule enable strict mutual TLS for the `httpbin.bar` workload:

```
$ cat <<EOF | kubectl apply -n bar -f -
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: "httpbin"
  namespace: "bar"
spec:
  selector:
    matchLabels:
      app: httpbin
  mtls:
    mode: STRICT
EOF
```

And a destination rule:

```
$ cat <<EOF | kubectl apply -n bar -f -  
apiVersion: networking.istio.io/v1alpha3  
kind: DestinationRule  
metadata:  
  name: "httpbin"  
spec:  
  host: "httpbin.bar.svc.cluster.local"  
  trafficPolicy:  
    tls:  
      mode: ISTIO_MUTUAL  
EOF
```

Again, run the probing command. As expected, request from sleep.legacy to httpbin.bar starts failing with the same reasons.

```
$ for from in "foo" "bar" "legacy"; do for to in "foo" "bar" "legacy"; do kubectl exec "$(kubectl get pod -l app=sleep -n ${from} -o jsonpath={.items..metadata.name})" -c sleep -n ${from} -- curl "http://httpbin.${to}:8000/ip" -s -o /dev/null -w "sleep.${from} to httpbin.${to}: %{http_code}\n"; done; done
sleep.foo to httpbin.foo: 200
sleep.foo to httpbin.bar: 200
sleep.foo to httpbin.legacy: 200
sleep.bar to httpbin.foo: 200
sleep.bar to httpbin.bar: 200
sleep.bar to httpbin.legacy: 200
sleep.legacy to httpbin.foo: 000
command terminated with exit code 56
sleep.legacy to httpbin.bar: 000
command terminated with exit code 56
sleep.legacy to httpbin.legacy: 200
```

```
...  
sleep.legacy to httpbin.bar: 000  
command terminated with exit code 56
```

To refine the mutual TLS settings per port, you must configure the `portLevelMtls` section. For example, the following peer authentication policy requires mutual TLS on all ports, except port 80:

```
$ cat <<EOF | kubectl apply -n bar -f -
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: "httpbin"
  namespace: "bar"
spec:
  selector:
    matchLabels:
      app: httpbin
  mtls:
    mode: STRICT
  portLevelMtls:
    80:
      mode: DISABLE
EOF
```

As before, you also need a destination rule:

```
$ cat <<EOF | kubectl apply -n bar -f -
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: "httpbin"
spec:
  host: httpbin.bar.svc.cluster.local
  trafficPolicy:
    tls:
      mode: ISTIO_MUTUAL
    portLevelSettings:
      - port:
          number: 8000
        tls:
          mode: DISABLE
EOF
```

1. The port value in the peer authentication policy is the container's port. The value the destination

rule is the service's port.

2. You can only use `portLevelMtls` if the port is bound to a service. Istio ignores it otherwise.

```
$ for from in "foo" "bar" "legacy"; do for to in "foo" "bar" "legacy"; do kubectl exec "$(kubectl get pod -l app=sleep -n ${from} -o jsonpath={.items..metadata.name})" -c sleep -n ${from} -- curl "http://httpbin.${to}:8000/ip" -s -o /dev/null -w "sleep.${from} to httpbin.${to}: %{http_code}\n"; done; done
sleep.foo to httpbin.foo: 200
sleep.foo to httpbin.bar: 200
sleep.foo to httpbin.legacy: 200
sleep.bar to httpbin.foo: 200
sleep.bar to httpbin.bar: 200
sleep.bar to httpbin.legacy: 200
sleep.legacy to httpbin.foo: 000
command terminated with exit code 56
sleep.legacy to httpbin.bar: 200
sleep.legacy to httpbin.legacy: 200
```

# Policy precedence

A workload-specific peer authentication policy takes precedence over a namespace-wide policy. You can test this behavior if you add a policy to disable mutual TLS for the `httpbin.foo` workload, for example. Note that you've already created a namespace-wide policy that enables mutual TLS for all services in namespace `foo` and observe that requests from `sleep.legacy` to `httpbin.foo` are failing (see above).

```
$ cat <<EOF | kubectl apply -n foo -f -
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: "overwrite-example"
  namespace: "foo"
spec:
  selector:
    matchLabels:
      app: httpbin
  mtls:
    mode: DISABLE
EOF
```

and destination rule:

```
$ cat <<EOF | kubectl apply -n foo -f -
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: "overwrite-example"
spec:
  host: httpbin.foo.svc.cluster.local
  trafficPolicy:
    tls:
      mode: DISABLE
EOF
```

Re-running the request from `sleep.legacy`, you should see a success return code again (200), confirming service-specific policy overrides the namespace-wide policy.

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n legacy -o json path={.items..metadata.name})" -c sleep -n legacy -- curl http://httpbin.foo:8000/ip -s -o /dev/null -w "%{http_code}\n"  
200
```

## Cleanup part 2

Remove policies and destination rules created in the above steps:

```
$ kubectl delete peerauthentication default overwrite-example -n foo  
$ kubectl delete peerauthentication httpbin -n bar  
$ kubectl delete destinationrules overwrite-example -n foo  
$ kubectl delete destinationrules httpbin -n bar
```

## End-user authentication

To experiment with this feature, you need a valid JWT. The JWT must correspond to the JWKS endpoint you want to use for the demo. This tutorial uses the test token JWT test and JWKS endpoint from the Istio

code base.

Also, for convenience, expose `httpbin.foo` via  
`ingressgateway` (for more details, see the `ingress` task).

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: httpbin-gateway
  namespace: foo
spec:
  selector:
    istio: ingressgateway # use Istio default gateway implementation
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"
EOF
```

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: httpbin
  namespace: foo
spec:
  hosts:
  - "*"
  gateways:
  - httpbin-gateway
  http:
  - route:
    - destination:
        port:
          number: 8000
        host: httpbin.foo.svc.cluster.local
EOF
```

Follow the instructions in Determining the ingress IP and ports to define the INGRESS\_HOST and INGRESS\_PORT environment variables.

And run a test query

```
$ curl "$INGRESS_HOST:$INGRESS_PORT/headers" -s -o /dev/null -w "%{http_code}\n"  
200
```

Now, add a request authentication policy that requires end-user JWT for the ingress gateway.

```
$ kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: RequestAuthentication
metadata:
  name: "jwt-example"
  namespace: istio-system
spec:
  selector:
    matchLabels:
      istio: ingressgateway
  jwtRules:
  - issuer: "testing@secure.istio.io"
    jwksUri: "https://raw.githubusercontent.com/istio/istio/release-1.11/security/tools/jwt/samples/jwks.json"
EOF
```

Apply the policy to the namespace of the workload it selects, `ingressgateway` in this case. The namespace

you need to specify is then `istio-system`.

If you provide a token in the authorization header, its implicitly default location, Istio validates the token using the public key set, and rejects requests if the bearer token is invalid. However, requests without tokens are accepted. To observe this behavior, retry the request without a token, with a bad token, and with a valid token:

```
$ curl "$INGRESS_HOST:$INGRESS_PORT/headers" -s -o /dev/null -w "%{http_code}\n"  
200
```

```
$ curl --header "Authorization: Bearer deadbeef" "$INGRESS_HOST:$INGRESS_PORT/headers" -s -o /dev/null -w "%{http_code}\n"  
401
```

```
$ TOKEN=$(curl https://raw.githubusercontent.com/istio/istio/release-1.11/security/tools/jwt/samples/demo.jwt -s)  
$ curl --header "Authorization: Bearer $TOKEN" "$INGRESS_HOST:$INGRESS_PORT/headers" -s -o /dev/null -w "%{http_code}\n"  
200
```

To observe other aspects of JWT validation, use the script `gen-jwt.py` to generate new tokens to test with different issuer, audiences, expiry date, etc. The script can be downloaded from the Istio repository:

```
$ wget --no-verbose https://raw.githubusercontent.com/istio/isti  
o/release-1.11/security/tools/jwt/samples/gen-jwt.py
```

You also need the key.pem file:

```
$ wget --no-verbose https://raw.githubusercontent.com/istio/isti  
o/release-1.11/security/tools/jwt/samples/key.pem
```

①

Download the jwcrypto library, if you haven't installed it on your system.

The JWT authentication has 60 seconds clock skew,

this means the JWT token will become valid 60 seconds earlier than its configured `nbf` and remain valid 60 seconds after its configured `exp`.

For example, the command below creates a token that expires in 5 seconds. As you see, Istio authenticates requests using that token successfully at first but rejects them after 65 seconds:

```
$ TOKEN=$(python3 ./gen-jwt.py ./key.pem --expire 5)
$ for i in $(seq 1 10); do curl --header "Authorization: Bearer
$TOKEN" "$INGRESS_HOST:$INGRESS_PORT/headers" -s -o /dev/null -w
"%{http_code}\n"; sleep 10; done
200
200
200
200
200
200
200
401
401
401
```

You can also add a JWT policy to an ingress gateway (e.g., service `istio-ingressgateway.istio-system.svc.cluster.local`). This is often used to define a

JWT policy for all services bound to the gateway, instead of for individual services.

## Require a valid token

To reject requests without valid tokens, add an authorization policy with a rule specifying a `DENY` action for requests without request principals, shown as `notRequestPrincipals: ["*"]` in the following example. Request principals are available only when valid JWT tokens are provided. The rule therefore denies requests without valid tokens.

```
$ kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: "frontend-ingress"
  namespace: istio-system
spec:
  selector:
    matchLabels:
      istio: ingressgateway
  action: DENY
  rules:
  - from:
    - source:
        notRequestPrincipals: ["*"]
EOF
```

Retry the request without a token. The request now

fails with error code 403:

```
$ curl "$INGRESS_HOST:$INGRESS_PORT/headers" -s -o /dev/null -w "%{http_code}\n"  
403
```

## Require valid tokens per-path

To refine authorization with a token requirement per host, path, or method, change the authorization policy to only require JWT on /headers. When this

authorization rule takes effect, requests to \$INGRESS\_HOST:\$INGRESS\_PORT/headers fail with the error code 403. Requests to all other paths succeed, for example \$INGRESS\_HOST:\$INGRESS\_PORT/ip.

```
$ kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: "frontend-ingress"
  namespace: istio-system
spec:
  selector:
    matchLabels:
      istio: ingressgateway
  action: DENY
  rules:
  - from:
    - source:
        notRequestPrincipals: ["*"]
  to:
  - operation:
      paths: ["/headers"]
EOF
```

```
$ curl "$INGRESS_HOST:$INGRESS_PORT/headers" -s -o /dev/null -w "%{http_code}\n"  
403
```

```
$ curl "$INGRESS_HOST:$INGRESS_PORT/ip" -s -o /dev/null -w "%{ht  
tp_code}\n"  
200
```

## Cleanup part 3

1. Remove authentication policy:

```
$ kubectl -n istio-system delete requestauthentication jwt-  
example
```

## 2. Remove authorization policy:

```
$ kubectl -n istio-system delete authorizationpolicy frontend-ingress
```

## 3. Remove the token generator script and key file:

```
$ rm -f ./gen-jwt.py ./key.pem
```

## 4. If you are not planning to explore any follow-on tasks, you can remove all resources simply by deleting test namespaces.

```
$ kubectl delete ns foo bar legacy
```

# Mutual TLS Migration

⌚ 4 minute read ✓ page test

---

This task shows how to ensure your workloads only communicate using mutual TLS as they are migrated to Istio.

Istio automatically configures workload sidecars to use mutual TLS when calling other workloads. By

default, Istio configures the destination workloads using PERMISSIVE mode. When PERMISSIVE mode is enabled, a service can accept both plain text and mutual TLS traffic. In order to only allow mutual TLS traffic, the configuration needs to be changed to STRICT mode.

You can use the Grafana dashboard to check which workloads are still sending plaintext traffic to the workloads in PERMISSIVE mode and choose to lock them down once the migration is done.

# Before you begin

- Understand Istio authentication policy and related mutual TLS authentication concepts.
- Read the authentication policy task to learn how to configure authentication policy.
- Have a Kubernetes cluster with Istio installed, without global mutual TLS enabled (for example, use the default configuration profile as described in installation steps).

In this task, you can try out the migration process by

creating sample workloads and modifying the policies to enforce STRICT mutual TLS between the workloads.

## Set up the cluster

- Create two namespaces, `foo` and `bar`, and deploy `httpbin` and `sleep` with sidecars on both of them:

```
$ kubectl create ns foo
$ kubectl apply -f <(istioctl kube-inject -f @samples/httpbin/httpbin.yaml@) -n foo
$ kubectl apply -f <(istioctl kube-inject -f @samples/sleep/sleep.yaml@) -n foo
$ kubectl create ns bar
$ kubectl apply -f <(istioctl kube-inject -f @samples/httpbin/httpbin.yaml@) -n bar
$ kubectl apply -f <(istioctl kube-inject -f @samples/sleep/sleep.yaml@) -n bar
```

- Create another namespace, legacy, and deploy sleep without a sidecar:

```
$ kubectl create ns legacy
$ kubectl apply -f @samples/sleep/sleep.yaml@ -n legacy
```

- Verify the setup by sending http requests (using

curl) from the sleep pods, in namespaces foo, bar and legacy, to httpbin.foo and httpbin.bar. All requests should succeed with return code 200.

```
$ for from in "foo" "bar" "legacy"; do for to in "foo" "bar"; do kubectl exec "$(kubectl get pod -l app=sleep -n ${from} -o jsonpath={.items..metadata.name})" -c sleep -n ${from} -- curl http://httpbin.${to}:8000/ip -s -o /dev/null -w "sleep.${from} to httpbin.${to}: %{http_code}\n"; done; done  
sleep.foo to httpbin.foo: 200  
sleep.foo to httpbin.bar: 200  
sleep.bar to httpbin.foo: 200  
sleep.bar to httpbin.bar: 200  
sleep.legacy to httpbin.foo: 200  
sleep.legacy to httpbin.bar: 200
```

If any of the curl commands fail, ensure that there are no existing authentication policies or destination rules that might interfere with requests to the httpbin service.



```
$ kubectl get peerauthentication --all-namespaces
```

```
No resources found
```

```
$ kubectl get destinationrule --all-namespaces
```

```
No resources found
```

# **Lock down to mutual TLS by namespace**

After migrating all clients to Istio and injecting the Envoy sidecar, you can lock down workloads in the `foo` namespace to only accept mutual TLS traffic.

```
$ kubectl apply -n foo -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: "default"
spec:
  mtls:
    mode: STRICT
EOF
```

Now, you should see the request from `sleep.legacy` to `httpbin.foo` failing.

```
$ for from in "foo" "bar" "legacy"; do for to in "foo" "bar"; do
  kubectl exec "$(kubectl get pod -l app=sleep -n ${from} -o json
path=.items..metadata.name)" -c sleep -n ${from} -- curl http:
//httpbin.${to}:8000/ip -s -o /dev/null -w "sleep.${from} to ht
pbin.${to}: %{http_code}\n"; done; done
sleep.foo to httpbin.foo: 200
sleep.foo to httpbin.bar: 200
sleep.bar to httpbin.foo: 200
sleep.bar to httpbin.bar: 200
sleep.legacy to httpbin.foo: 000
command terminated with exit code 56
sleep.legacy to httpbin.bar: 200
```

If you installed Istio with

values.global.proxy.privileged=true, you can use tcpdump  
to verify traffic is encrypted or not.

```
$ kubectl exec -nfoo "$(kubectl get pod -nfoo -lapp=httpbin -ojs onpath={.items..metadata.name})" -c istio-proxy -- sudo tcpdump dst port 80 -A
tcpdump: verbose output suppressed, use -v or -vv for full proto
col decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262
144 bytes
```

You will see plain text and encrypted text in the output when requests are sent from `sleep.legacy` and `sleep.foo` respectively.

If you can't migrate all your services to Istio (i.e., inject Envoy sidecar in all of them), you will need to continue to use `PERMISSIVE` mode. However, when configured with `PERMISSIVE` mode, no authentication or

authorization checks will be performed for plaintext traffic by default. We recommend you use Istio Authorization to configure different paths with different authorization policies.

## **Lock down mutual TLS for the entire mesh**

```
$ kubectl apply -n istio-system -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: "default"
spec:
  mTLS:
    mode: STRICT
EOF
```

Now, both the `foo` and `bar` namespaces enforce mutual TLS only traffic, so you should see requests from `sleep.legacy` failing for both.

```
$ for from in "foo" "bar" "legacy"; do for to in "foo" "bar"; do
  kubectl exec "$(kubectl get pod -l app=sleep -n ${from} -o json
path=.items..metadata.name)" -c sleep -n ${from} -- curl http:
//httpbin.${to}:8000/ip -s -o /dev/null -w "sleep.${from} to ht
pbin.${to}: %{http_code}\n"; done; done
```

## Clean up the example

1. Remove the mesh-wide authentication policy.

```
$ kubectl delete peerauthentication -n istio-system default
```

1. Remove the test namespaces.

```
$ kubectl delete ns foo bar legacy  
Namespaces foo bar legacy deleted.
```



# HTTP Traffic

⌚ 4 minute read ✓ page test

---

This task shows you how to set up Istio authorization policy of `ALLOW` action for HTTP traffic in an Istio mesh.

## Before you begin

Before you begin this task, do the following:

- Read the Istio authorization concepts.
- Follow the Istio installation guide to install Istio with mutual TLS enabled.
- Deploy the Bookinfo sample application.

After deploying the Bookinfo application, go to the Bookinfo product page at

`http://$GATEWAY_URL/productpage`. On the product page, you can see the following sections:

- **Book Details** on the lower left side, which

includes: book type, number of pages, publisher, etc.

- **Book Reviews** on the lower right of the page.

When you refresh the page, the app shows different versions of reviews in the product page. The app presents the reviews in a round robin style: red stars, black stars, or no stars.

If you don't see the expected output in the browser as you follow the task, retry in a few more seconds because some delay is possible

due to caching and other propagation overhead.

This task requires mutual TLS enabled because the following examples use principal and namespace in the policies.

## Configure access control

# for workloads using HTTP traffic

Using Istio, you can easily setup access control for workloads in your mesh. This task shows you how to set up access control using Istio authorization. First, you configure a simple `allow-nothing` policy that rejects all requests to the workload, and then grant more access to the workload gradually and incrementally.

1. Run the following command to create a `allow-nothing` policy in the `default` namespace. The policy

doesn't have a `selector` field, which applies the policy to every workload in the `default` namespace. The `spec:` field of the policy has the empty value `{}`. That value means that no traffic is permitted, effectively denying all requests.

```
$ kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-nothing
  namespace: default
spec:
  {}
EOF
```

Point your browser at the Bookinfo productpage

(`http://$GATEWAY_URL/productpage`). You should see "RBAC: access denied". The error shows that the configured deny-all policy is working as intended, and Istio doesn't have any rules that allow any access to workloads in the mesh.

2. Run the following command to create a `productpage-viewer` policy to allow access with `GET` method to the `productpage` workload. The policy does not set the `from` field in the rules which means all sources are allowed, effectively allowing all users and workloads:

```
$ kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: "productpage-viewer"
  namespace: default
spec:
  selector:
    matchLabels:
      app: productpage
  action: ALLOW
  rules:
  - to:
    - operation:
        methods: ["GET"]
EOF
```

Point your browser at the Bookinfo productpage ([http://\\$GATEWAY\\_URL/productpage](http://$GATEWAY_URL/productpage)). Now you should

see the “Bookinfo Sample” page. However, you can see the following errors on the page:

- Error fetching product details
- Error fetching product reviews on the page.

These errors are expected because we have not granted the `productpage` workload access to the `details` and `reviews` workloads. Next, you need to configure a policy to grant access to those workloads.

3. Run the following command to create the `details-viewer` policy to allow the `productpage` workload, which issues requests using the

cluster.local/ns/default/sa/bookinfo-productpage  
service account, to access the details workload  
through GET methods:

```
$ kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: "details-viewer"
  namespace: default
spec:
  selector:
    matchLabels:
      app: details
  action: ALLOW
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/default/sa/bookinfo-
productpage"]
    to:
    - operation:
        methods: ["GET"]
EOF
```

4. Run the following command to create a policy reviews-viewer to allow the productpage workload, which issues requests using the cluster.local/ns/default/sa/bookinfo-productpage service account, to access the reviews workload through GET methods:

```
$ kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: "reviews-viewer"
  namespace: default
spec:
  selector:
    matchLabels:
      app: reviews
```

```
action: ALLOW
rules:
- from:
  - source:
    principals: ["cluster.local/ns/default/sa/bookinfo-
productpage"]
  to:
- operation:
  methods: ["GET"]
EOF
```

Point your browser at the Bookinfo productpage ([http://\\$GATEWAY\\_URL/productpage](http://$GATEWAY_URL/productpage)). Now, you should see the “Bookinfo Sample” page with “Book Details” on the lower left part, and “Book Reviews” on the lower right part. However, in the “Book Reviews” section, there is an error Ratings

service currently unavailable.

This is because the reviews workload doesn't have permission to access the ratings workload. To fix this issue, you need to grant the reviews workload access to the ratings workload. Next, we configure a policy to grant the reviews workload that access.

5. Run the following command to create the ratings-viewer policy to allow the reviews workload, which issues requests using the cluster.local/ns/default/sa/bookinfo-reviews service account, to access the ratings workload through GET methods:

```
$ kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: "ratings-viewer"
  namespace: default
spec:
  selector:
    matchLabels:
      app: ratings
  action: ALLOW
  rules:
    - from:
        - source:
            principals: ["cluster.local/ns/default/sa/bookinfo-
reviews"]
        to:
          - operation:
              methods: ["GET"]
EOF
```

Point your browser at the Bookinfo productpage ([http://\\$GATEWAY\\_URL/productpage](http://$GATEWAY_URL/productpage)). You should see the “black” and “red” ratings in the “Book Reviews” section.

**Congratulations!** You successfully applied authorization policy to enforce access control for workloads using HTTP traffic.

## Clean up

1. Remove all authorization policies from your

## configuration:

```
$ kubectl delete authorizationpolicy.security.istio.io/allow-nothing
$ kubectl delete authorizationpolicy.security.istio.io/productpage-viewer
$ kubectl delete authorizationpolicy.security.istio.io/details-viewer
$ kubectl delete authorizationpolicy.security.istio.io/reviews-viewer
$ kubectl delete authorizationpolicy.security.istio.io/ratings-viewer
```

# TCP Traffic

⌚ 6 minute read ✓ page test

---

This task shows you how to set up Istio authorization policy for TCP traffic in an Istio mesh.

## Before you begin

Before you begin this task, do the following:

- Read the Istio authorization concepts.
- Install Istio using the Istio installation guide.
- Deploy two workloads named `sleep` and `tcp-echo` together in a namespace, for example `foo`. Both workloads run with an Envoy proxy in front of each. The `tcp-echo` workload listens on port 9000, 9001 and 9002 and echoes back any traffic it received with a prefix `hello`. For example, if you send “world” to `tcp-echo`, it will reply with `hello world`. The `tcp-echo` Kubernetes service object only declares the ports 9000 and 9001, and omits the

port 9002. A pass-through filter chain will handle port 9002 traffic. Deploy the example namespace and workloads using the following command:

```
$ kubectl create ns foo
$ kubectl apply -f <(istioctl kube-inject -f @samples/tcp-echo/tcp-echo.yaml@) -n foo
$ kubectl apply -f <(istioctl kube-inject -f @samples/sleep/sleep.yaml@) -n foo
```

- Verify that sleep successfully communicates with tcp-echo on ports 9000 and 9001 using the following command:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o js  
onpath={.items..metadata.name})" -c sleep -n foo -- sh -c '  
echo "port 9000" | nc tcp-echo 9000' | grep "hello" && echo  
'connection succeeded' || echo 'connection rejected'  
hello port 9000  
connection succeeded
```

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o js  
onpath={.items..metadata.name})" -c sleep -n foo -- sh -c '  
echo "port 9001" | nc tcp-echo 9001' | grep "hello" && echo  
'connection succeeded' || echo 'connection rejected'  
hello port 9001  
connection succeeded
```

- Verify that sleep successfully communicates with tcp-echo on port 9002. You need to send the traffic directly to the pod IP of tcp-echo because the port

9002 is not defined in the Kubernetes service object of `tcp-echo`. Get the pod IP address and send the request with the following command:

```
$ TCP_ECHO_IP=$(kubectl get pod "$(
    kubectl get pod -l app=tcp-echo -n foo -o jsonpath={.items..metadata.name}
)" -n foo -o jsonpath=".status.podIP")"
$ kubectl exec "$(
    kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name}
)" -c sleep -n foo -- sh -c "
echo \"port 9002\" | nc $TCP_ECHO_IP 9002" | grep "hello" &
& echo 'connection succeeded' || echo 'connection rejected'
hello port 9002
connection succeeded
```

If you don't see the expected output, retry after a few seconds. Caching and

propagation can cause a delay.

## Configure access control for a TCP workload

1. Create the `tcp-policy` authorization policy for the `tcp-echo` workload in the `foo` namespace. Run the following command to apply the policy to allow requests to port 9000 and 9001:

```
$ kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: tcp-policy
  namespace: foo
spec:
  selector:
    matchLabels:
      app: tcp-echo
  action: ALLOW
  rules:
  - to:
    - operation:
        ports: ["9000", "9001"]
EOF
```

2. Verify that requests to port 9000 are allowed using the following command:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o js onpath=.items..metadata.name)" -c sleep -n foo -- sh -c 'echo "port 9000" | nc tcp-echo 9000' | grep "hello" && echo 'connection succeeded' || echo 'connection rejected'  
hello port 9000  
connection succeeded
```

3. Verify that requests to port 9001 are allowed using the following command:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o js onpath=.items..metadata.name)" -c sleep -n foo -- sh -c 'echo "port 9001" | nc tcp-echo 9001' | grep "hello" && echo 'connection succeeded' || echo 'connection rejected'  
hello port 9001  
connection succeeded
```

4. Verify that requests to port 9002 are denied. This

is enforced by the authorization policy which also applies to the pass through filter chain, even if the port is not declared explicitly in the `tcp-echo` Kubernetes service object. Run the following command and verify the output:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})" -c sleep -n foo -- sh -c "echo \"port 9002\" | nc $TCP_ECHO_IP 9002" | grep "hello" & & echo 'connection succeeded' || echo 'connection rejected'
```

connection rejected

5. Update the policy to add an HTTP-only field named `methods` for port 9000 using the following command:

```
$ kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: tcp-policy
  namespace: foo
spec:
  selector:
    matchLabels:
      app: tcp-echo
  action: ALLOW
  rules:
  - to:
    - operation:
        methods: ["GET"]
        ports: ["9000"]
EOF
```

6. Verify that requests to port 9000 are denied. This

occurs because the rule becomes invalid when it uses an HTTP-only field (`methods`) for TCP traffic. Istio ignores the invalid ALLOW rule. The final result is that the request is rejected, because it does not match any ALLOW rules. Run the following command and verify the output:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})" -c sleep -n foo -- sh -c 'echo "port 9000" | nc tcp-echo 9000' | grep "hello" && echo 'connection succeeded' || echo 'connection rejected'  
connection rejected
```

7. Verify that requests to port 9001 are denied. This occurs because the requests do not match any

ALLOW rules. Run the following command and verify the output:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o js onpath={.items..metadata.name})" -c sleep -n foo -- sh -c 'echo "port 9001" | nc tcp-echo 9001' | grep "hello" && echo 'connection succeeded' || echo 'connection rejected'  
connection rejected
```

8. Update the policy to a DENY policy using the following command:

```
$ kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: tcp-policy
  namespace: foo
spec:
  selector:
    matchLabels:
      app: tcp-echo
  action: DENY
  rules:
  - to:
    - operation:
        methods: ["GET"]
        ports: ["9000"]
EOF
```

9. Verify that requests to port 9000 are denied. This

occurs because Istio ignores the HTTP-only fields in an invalid DENY rule. This is different from an invalid ALLOW rule, which causes Istio to ignore the entire rule. The final result is that only the ports field is used by Istio and the requests are denied because they match with the ports:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})" -c sleep -n foo -- sh -c 'echo "port 9000" | nc tcp-echo 9000' | grep "hello" && echo 'connection succeeded' || echo 'connection rejected'  
connection rejected
```

10. Verify that requests to port 9001 are allowed. This occurs because the requests do not match

the ports in the DENY policy:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})" -c sleep -n foo -- sh -c 'echo "port 9001" | nc tcp-echo 9001' | grep "hello" && echo 'connection succeeded' || echo 'connection rejected'  
hello port 9001  
connection succeeded
```

## Clean up

1. Remove the namespace foo:

```
$ kubectl delete namespace foo
```

# JWT Token

⌚ 4 minute read ✓ page test

---

This task shows you how to set up an Istio authorization policy to enforce access based on a JSON Web Token (JWT). An Istio authorization policy supports both string typed and list-of-string typed JWT claims.

# Before you begin

Before you begin this task, do the following:

- Complete the Istio end user authentication task.
- Read the Istio authorization concepts.
- Install Istio using Istio installation guide.
- Deploy two workloads: `httpbin` and `sleep`. Deploy these in one namespace, for example `foo`. Both workloads run with an Envoy proxy in front of each. Deploy the example namespace and workloads using these commands:

```
$ kubectl create ns foo  
$ kubectl apply -f <(istioctl kube-inject -f @samples/httpbin/httpbin.yaml@) -n foo  
$ kubectl apply -f <(istioctl kube-inject -f @samples/sleep/sleep.yaml@) -n foo
```

- Verify that sleep successfully communicates with httpbin using this command:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})" -c sleep -n foo -- curl http://httpbin.foo:8000/ip -sS -o /dev/null -w "%{http_code}\n"  
200
```

If you don't see the expected output, retry

after a few seconds. Caching and propagation can cause a delay.

## **Allow requests with valid JWT and list-typed claims**

1. The following command creates the `jwt-example` request authentication policy for the `httpbin` workload in the `foo` namespace. This policy for `httpbin` workload accepts a JWT issued by

testing@secure.istio.io:

```
$ kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: RequestAuthentication
metadata:
  name: "jwt-example"
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
  jwtRules:
    - issuer: "testing@secure.istio.io"
      jwksUri: "https://raw.githubusercontent.com/istio/istio
/release-1.11/security/tools/jwt/samples/jwks.json"
EOF
```

2. Verify that a request with an invalid JWT is

denied:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})" -c sleep -n foo -- curl "http://httpbin.foo:8000/headers" -sS -o /dev/null -H "Authorization: Bearer invalidToken" -w "%{http_code}\n"  
401
```

3. Verify that a request without a JWT is allowed because there is no authorization policy:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})" -c sleep -n foo -- curl "http://httpbin.foo:8000/headers" -sS -o /dev/null -w "%{http_code}\n"  
200
```

4. The following command creates the require-jwt

authorization policy for the `httpbin` workload in the `foo` namespace. The policy requires all requests to the `httpbin` workload to have a valid JWT with `requestPrincipal` set to

`testing@secure.istio.io/testing@secure.istio.io`.

Istio constructs the `requestPrincipal` by combining the `iss` and `sub` of the JWT token with a `/` separator as shown:

```
$ kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: require-jwt
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
  action: ALLOW
  rules:
  - from:
    - source:
        requestPrincipals: ["testing@secure.istio.io/testing
@secure.istio.io"]
EOF
```

5. Get the JWT that sets the `iss` and `sub` keys to the

same value, testing@secure.istio.io. This causes Istio to generate the attribute requestPrincipal with the value

testing@secure.istio.io/testing@secure.istio.io:

```
$ TOKEN=$(curl https://raw.githubusercontent.com/istio/istio/release-1.11/security/tools/jwt/samples/demo.jwt -s) && echo "$TOKEN" | cut -d '.' -f2 - | base64 --decode - {"exp":4685989700,"foo":"bar","iat":1532389700,"iss":"testing@secure.istio.io","sub":"testing@secure.istio.io"}
```

6. Verify that a request with a valid JWT is allowed:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o js  
onpath={.items..metadata.name})" -c sleep -n foo -- curl "h  
ttp://httpbin.foo:8000/headers" -sS -o /dev/null -H "Author  
ization: Bearer $TOKEN" -w "%{http_code}\n"  
200
```

## 7. Verify that a request without a JWT is denied:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o js  
onpath={.items..metadata.name})" -c sleep -n foo -- curl "h  
ttp://httpbin.foo:8000/headers" -sS -o /dev/null -w "%{http  
_code}\n"  
403
```

## 8. The following command updates the require-jwt authorization policy to also require the JWT to have a claim named groups containing the value

group1:

```
$ kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: require-jwt
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
  action: ALLOW
  rules:
    - from:
        - source:
            requestPrincipals: ["testing@secure.istio.io/testing
@secure.istio.io"]
            when:
              - key: request.auth.claims[groups]
                values: ["group1"]
EOF
```



Don't include quotes in the request.auth.claims field unless the claim itself has quotes in it.

9. Get the JWT that sets the groups claim to a list of strings: group1 and group2:

```
$ TOKEN_GROUP=$(curl https://raw.githubusercontent.com/istio/istio/release-1.11/security/tools/jwt/samples/groups-scopes.jwt -s) && echo "$TOKEN_GROUP" | cut -d '.' -f2 - | base64 --decode -  
{"exp":3537391104,"groups":["group1","group2"],"iat":1537391104,"iss":"testing@secure.istio.io","scope":["scope1","scope2"],"sub":"testing@secure.istio.io"}
```

10. Verify that a request with the JWT that includes group1 in the groups claim is allowed:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})" -c sleep -n foo -- curl "http://httpbin.foo:8000/headers" -sS -o /dev/null -H "Authorization: Bearer $TOKEN_GROUP" -w "%{http_code}\n"  
200
```

11. Verify that a request with a JWT, which doesn't have the groups claim is rejected:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})" -c sleep -n foo -- curl "http://httpbin.foo:8000/headers" -sS -o /dev/null -H "Authorization: Bearer $TOKEN" -w "%{http_code}\n"  
403
```

# Clean up

1. Remove the namespace foo:

```
$ kubectl delete namespace foo
```

# External Authorization

⌚ 7 minute read ✓ page test

---

This task shows you how to set up an Istio authorization policy using a new experimental value for the `action` field, `CUSTOM`, to delegate the access control to an external authorization system. This can be used to integrate with OPA authorization, oauth2-

proxy, your own custom external authorization server and more.



The following information describes an experimental feature, which is intended for evaluation purposes only.

## Before you begin

Before you begin this task, do the following:

- Read the Istio authorization concepts.
- Follow the Istio installation guide to install Istio.
- Deploy test workloads:

This task uses two workloads, `httpbin` and `sleep`, both deployed in namespace `foo`. Both workloads run with an Envoy proxy sidecar. Deploy the `foo` namespace and workloads with the following command:

```
$ kubectl create ns foo  
$ kubectl label ns foo istio-injection=enabled  
$ kubectl apply -f @samples/httpbin/httpbin.yaml@ -n foo  
$ kubectl apply -f @samples/sleep/sleep.yaml@ -n foo
```

- Verify that sleep can access httpbin with the following command:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})" -c sleep -n foo -- curl http://httpbin.foo:8000/ip -s -o /dev/null -w "%{http_code}\n"  
200
```

If you don't see the expected output as you follow the task, retry after a few seconds.

Caching and propagation overhead can cause some delay.

## **Deploy the external authorizer**

First, you need to deploy the external authorizer. For this, you will simply deploy the sample external authorizer in a standalone pod in the mesh.

1. Run the following command to deploy the sample external authorizer:

```
$ kubectl apply -n foo -f https://raw.githubusercontent.com  
/istio/istio/release-1.11/samples/extauthz/ext-authz.yaml  
service/ext-authz created  
deployment.apps/ext-authz created
```

2. Verify the sample external authorizer is up and running:

```
$ kubectl logs "$(kubectl get pod -l app=ext-authz -n foo -  
o jsonpath={.items..metadata.name})" -n foo -c ext-authz  
2021/01/07 22:55:47 Starting HTTP server at [::]:8000  
2021/01/07 22:55:47 Starting gRPC server at [::]:9000
```

Alternatively, you can also deploy the external

authorizer as a separate container in the same pod of the application that needs the external authorization or even deploy it outside of the mesh. In either case, you will also need to create a service entry resource to register the service to the mesh and make sure it is accessible to the proxy.

The following is an example service entry for an external authorizer deployed in a separate container in the same pod of the application that needs the external authorization.

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: external-authz-grpc-local
spec:
  hosts:
    - "external-authz-grpc.local" # The service name to be used in
the extension provider in the mesh config.
  endpoints:
    - address: "127.0.0.1"
  ports:
    - name: grpc
      number: 9191 # The port number to be used in the extension p
rovider in the mesh config.
      protocol: GRPC
  resolution: STATIC
```

# Define the external authorizer

In order to use the `CUSTOM` action in the authorization policy, you must then define the external authorizer that is allowed to be used in the mesh. This is currently defined in the extension provider in the mesh config.

Currently, the only supported extension provider type is the Envoy `ext_authz` provider. The external authorizer must implement the corresponding Envoy `ext_authz`

check API.

In this task, you will use a sample external authorizer which allows requests with the header `x-ext-authz: allow`.

1. Edit the mesh config with the following command:

```
$ kubectl edit configmap istio -n istio-system
```

2. In the editor, add the extension provider definitions shown below:

The following content defines two external providers `sample-ext-authz-grpc` and `sample-ext-`

authz-`http` using the same service `ext-authz.foo.svc.cluster.local`. The service implements both the HTTP and gRPC check API as defined by the Envoy `ext_authz` filter. You will deploy the service in the following step.

```
data:  
  mesh: |-  
    # Add the following content to define the external auth  
    orizers.  
    extensionProviders:  
      - name: "sample-ext-authz-grpc"  
        envoyExtAuthzGrpc:  
          service: "ext-authz.foo.svc.cluster.local"  
          port: "9000"  
      - name: "sample-ext-authz-http"  
        envoyExtAuthzHttp:  
          service: "ext-authz.foo.svc.cluster.local"  
          port: "8000"  
          includeHeadersInCheck: ["x-ext-authz"]
```

Alternatively, you can modify the extension provider to control the behavior of the `ext_authz` filter for things like what headers to send to the

external authorizer, what headers to send to the application backend, the status to return on error and more. For example, the following defines an extension provider that can be used with the oauth2-proxy:

```
data:  
  mesh: |-  
    extensionProviders:  
      - name: "oauth2-proxy"  
    envoyExtAuthzHttp:  
      service: "oauth2-proxy.foo.svc.cluster.local"  
      port: "4180" # The default port used by oauth2-prox  
y.  
      includeHeadersInCheck: ["authorization", "cookie"]  
# headers sent to the oauth2-proxy in the check request.  
      headersToUpstreamOnAllow: ["authorization", "path",  
        "x-auth-request-user", "x-auth-request-email", "x-auth-req  
uest-access-token"] # headers sent to backend application w  
hen request is allowed.  
      headersToDownstreamOnDeny: ["content-type", "set-co  
okie"] # headers sent back to the client when request is de  
nied.
```

### 3. Restart Istiod to allow the change to take effect

with the following command:

```
$ kubectl rollout restart deployment/istiod -n istio-system  
deployment.apps/istiod restarted
```

## Enable with external authorization

The external authorizer is now ready to be used by the authorization policy.

1. Enable the external authorization with the following command:

The following command applies an authorization policy with the `CUSTOM` action value for the `httpbin` workload. The policy enables the external authorization for requests to path `/headers` using the external authorizer defined by `sample-ext-authz-grpc`.

```
$ kubectl apply -n foo -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: ext-authz
spec:
  selector:
```

```
matchLabels:  
    app: httpbin  
action: CUSTOM  
provider:  
    # The provider name must match the extension provider defined in the mesh config.  
    # You can also replace this with sample-ext-authz-http to test the other external authorizer definition.  
    name: sample-ext-authz-grpc  
rules:  
    # The rules specify when to trigger the external authorizer.  
    - to:  
        - operation:  
            paths: ["/headers"]  
EOF
```

At runtime, requests to path /headers of the httpbin workload will be paused by the ext\_authz filter,

and a check request will be sent to the external authorizer to decide whether the request should be allowed or denied.

2. Verify a request to path /headers with header x-ext-authz: deny is denied by the sample ext\_authz server:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})" -c sleep -n foo -- curl "http://httpbin.foo:8000/headers" -H "x-ext-authz: deny" -s  
denied by ext_authz for not found header `x-ext-authz: allow` in the request
```

3. Verify a request to path /headers with header x-ext-authz: allow is allowed by the sample ext\_authz

## server:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})" -c sleep -n foo -- curl "http://httpbin.foo:8000/headers" -H "x-ext-authz: allow" -s
{
  "headers": {
    "Accept": "*/*",
    "Host": "httpbin:8000",
    "User-Agent": "curl/7.76.0-DEV",
    "X-B3-Parentspanid": "430f770aeb7ef215",
    "X-B3-Sampled": "0",
    "X-B3-Spanid": "60ff95c5acdf5288",
    "X-B3-Traceid": "fba72bb5765daf5a430f770aeb7ef215",
    "X-Envoy-Attempt-Count": "1",
    "X-Ext-Authz": "allow",
    "X-Ext-Authz-Check-Result": "allowed",
    "X-Forwarded-Client-Cert": "By=spiffe://cluster.local/n
s/foo/sa/httpbin;Hash=e5178ee79066bfbaf
b1d98044fc
d0cf80db76
be8714c7a4b630c7922df520bf2;Subject=\\"\\";
URI=spiffe://clust
```

```
er.local/ns/foo/sa/sleep"  
}  
}  
}
```

4. Verify a request to path /ip is allowed and does not trigger the external authorization:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})" -c sleep -n foo -- curl "http://httpbin.foo:8000/ip" -s -o /dev/null -w "%{http_code}\n"  
200
```

5. Check the log of the sample ext\_authz server to confirm it was called twice (for the two requests). The first one was allowed and the second one was denied:

```
$ kubectl logs "$(kubectl get pod -l app=ext-authz -n foo -o jsonpath={.items..metadata.name})" -n foo -c ext-authz
2021/01/07 22:55:47 Starting HTTP server at [::]:8000
2021/01/07 22:55:47 Starting gRPC server at [::]:9000
2021/01/08 03:25:00 [gRPCv3][denied]: httpbin.foo:8000/headers, attributes: source:{address:{socket_address:{address:"10.44.0.22" port_value:52088}} principal:"spiffe://cluster.local/ns/foo/sa/sleep"} destination:{address:{socket_address:{address:"10.44.3.30" port_value:80}} principal:"spiffe://cluster.local/ns/foo/sa/httpbin"} request:{time:{seconds:1610076306 nanos:473835000} http:{id:"13869142855783664817" method:"GET" headers:{key:":authority" value:"httpbin.foo:8000"} headers:{key:"method" value:"GET"} headers:{key:"path" value:"/headers"} headers:{key:"accept" value:"*/*"} headers:{key:"content-length" value:"0"} headers:{key:"user-agent" value:"curl/7.74.0-DEV"} headers:{key:"x-b3-sampled" value:"1"} headers:{key:"x-b3-spanid" value:"377ba0cdc2334270"} headers:{key:"x-b3-traceid" value:"635187cb20d92f62377ba0cdc2334270"} headers:{key:"x-envoy-attempt-count" value:"1"} headers:{key:"x-ext-auth
```

```
z" value:"deny"} headers:{key:"x-forwarded-client-cert" value:"By=spiffe://cluster.local/ns/foo/sa/httpbin;Hash=dd14782fa2f439724d271dbed846ef843ff40d3932b615da650d028db655fc8d;Subject=\"\";URI=spiffe://cluster.local/ns/foo/sa/sleep"} headers:{key:"x-forwarded-proto" value:"http"} headers:{key:"x-request-id" value:"9609691a-4e9b-9545-ac71-3889bc2dffb0"} path:"/headers" host:"httpbin.foo:8000" protocol:"HTTP/1.1"}} metadata_context:{}
```

```
2021/01/08 03:25:06 [gRPCv3][allowed]: httpbin.foo:8000/headers, attributes: source:{address:{socket_address:{address:"10.44.0.22" port_value:52184}}} principal:"spiffe://cluster.local/ns/foo/sa/sleep"} destination:{address:{socket_address:{address:"10.44.3.30" port_value:80}}} principal:"spiffe://cluster.local/ns/foo/sa/httpbin"} request:{time:{seconds:1610076300 nanos:925912000} http:{id:"17995949296433813435" method:"GET" headers:{key)::authority" value:"httpbin.foo:8000"} headers:{key)::method" value:"GET"} headers:{key)::path" value:"/headers"} headers:{key:"accept" value:"*/*"} headers:{key:"content-length" value:"0"} headers:{key:"user-agent" value:"curl/7.74.0-DEV"} heade
```

```
rs:{key:"x-b3-sampled" value:"1"} headers:{key:"x-b3-span  
id" value:"a66b5470e922fa80"} headers:{key:"x-b3-traceid"  
value:"300c2f2b90a618c8a66b5470e922fa80"} headers:{key:"  
x-envoy-attempt-count" value:"1"} headers:{key:"x-ext-aut  
hz" value:"allow"} headers:{key:"x-forwarded-client-cert"  
value:"By=spiffe://cluster.local/ns/foo/sa/httpbin;Hash=d  
d14782fa2f439724d271dbed846ef843ff40d3932b615da650d028db655  
fc8d;Subject=\"\";URI=spiffe://cluster.local/ns/foo/sa/slee  
p"} headers:{key:"x-forwarded-proto" value:"http"} heade  
rs:{key:"x-request-id" value:"2b62daf1-00b9-97d9-91b8-ba61  
94ef58a4"} path:"/headers" host:"httpbin.foo:8000" proto  
col:"HTTP/1.1"}} metadata_context:{}
```

You can also tell from the log that mTLS is enabled for the connection between the ext-authz filter and the sample ext-authz server because the source principal is populated with the value spiffe://cluster.local/ns/foo/sa/sleep.

You can now apply another authorization policy for the sample `ext-authz` server to control who is allowed to access it.

## Clean up

1. Remove the namespace `foo` from your configuration:

```
$ kubectl delete namespace foo
```

2. Remove the extension provider definition from

the mesh config.

# Explicit Deny

⌚ 4 minute read ✓ page test

---

This task shows you how to set up Istio authorization policy of `DENY` action to explicitly deny traffic in an Istio mesh. This is different from the `ALLOW` action because the `DENY` action has higher priority and will not be bypassed by any `ALLOW` actions.

# Before you begin

Before you begin this task, do the following:

- Read the Istio authorization concepts.
- Follow the Istio installation guide [to install Istio](#).
- Deploy workloads:

This task uses two workloads, httpbin and sleep, deployed on one namespace, foo. Both workloads run with an Envoy proxy in front of each. Deploy the example namespace and workloads with the following command:

```
$ kubectl create ns foo  
$ kubectl apply -f <(istioctl kube-inject -f @samples/httpbin/httpbin.yaml@) -n foo  
$ kubectl apply -f <(istioctl kube-inject -f @samples/sleep/sleep.yaml@) -n foo
```

- Verify that sleep talks to httpbin with the following command:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath='{.items..metadata.name}')" -c sleep -n foo -- curl http://httpbin.foo:8000/ip -sS -o /dev/null -w "%{http_code}\n"  
200
```

If you don't see the expected output as you follow the task, retry after a few seconds.

Caching and propagation overhead can cause some delay.

## Explicitly deny a request

1. The following command creates the `deny-method-get` authorization policy for the `httpbin` workload in the `foo` namespace. The policy sets the `action` to `DENY` to deny requests that satisfy the conditions set in the `rules` section. This type of policy is

better known as deny policy. In this case, the policy denies requests if their method is GET.

```
$ kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: deny-method-get
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
  action: DENY
  rules:
  - to:
    - operation:
        methods: ["GET"]
EOF
```

## 2. Verify that GET requests are denied:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})" -c sleep -n foo -- curl "http://httpbin.foo:8000/get" -X GET -sS -o /dev/null -w "%{http_code}\n"  
403
```

## 3. Verify that POST requests are allowed:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})" -c sleep -n foo -- curl "http://httpbin.foo:8000/post" -X POST -sS -o /dev/null -w "%{http_code}\n"  
200
```

## 4. Update the deny-method-get authorization policy to deny GET requests only if the value of the HTTP

header `x-token` value is not `admin`. The following example policy sets the value of the `notValues` field to `["admin"]` to deny requests with a header value that is not `admin`:

```
$ kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: deny-method-get
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
  action: DENY
  rules:
  - to:
    - operation:
        methods: ["GET"]
    when:
    - key: request.headers[x-token]
      notValues: ["admin"]
EOF
```

5. Verify that GET requests with the HTTP header x-token: admin **are allowed**:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})" -c sleep -n foo -- curl "http://httpbin.foo:8000/get" -X GET -H "x-token: admin" -sS -o /dev/null -w "%{http_code}\n"  
200
```

6. Verify that GET requests with the HTTP header x-token: guest **are denied**:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})" -c sleep -n foo -- curl "http://httpbin.foo:8000/get" -X GET -H "x-token: guest" -sS -o /dev/null -w "%{http_code}\n"  
403
```

7. The following command creates the `allow-path-ip` authorization policy to allow requests at the `/ip` path to the `httpbin` workload. This authorization policy sets the `action` field to `ALLOW`. This type of policy is better known as an allow policy.

```
$ kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-path-ip
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
  action: ALLOW
  rules:
  - to:
    - operation:
        paths: ["/ip"]
EOF
```

8. Verify that GET requests with the HTTP header x-token: guest at path /ip are denied by the deny-

method-get policy. Deny policies takes precedence over the allow policies:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o js  
onpath={.items..metadata.name})" -c sleep -n foo -- curl "h  
ttp://httpbin.foo:8000/ip" -X GET -H "x-token: guest" -s -o  
/dev/null -w "%{http_code}\n"  
403
```

9. Verify that GET requests with the HTTP header x-token: admin at path /ip are allowed by the allow-path-ip policy:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o js onpath=.items..metadata.name)" -c sleep -n foo -- curl "http://httpbin.foo:8000/ip" -X GET -H "x-token: admin" -s -o /dev/null -w "%{http_code}\n"
```

200

0. Verify that GET requests with the HTTP header x-token: admin at path /get are denied because they don't match the allow-path-ip policy:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o js onpath=.items..metadata.name)" -c sleep -n foo -- curl "http://httpbin.foo:8000/get" -X GET -H "x-token: admin" -s -o /dev/null -w "%{http_code}\n"
```

403

# Clean up

1. Remove the namespace foo from your configuration:

```
$ kubectl delete namespace foo
```

# Ingress Gateway

⌚ 8 minute read ✓ page test

---

This task shows you how to enforce IP-based access control on an Istio ingress gateway using an authorization policy.

## Before you begin

Before you begin this task, do the following:

- Read the Istio authorization concepts.
- Install Istio using the Istio installation guide.
- Deploy a workload, `httpbin` in a namespace, for example `foo`, and expose it through the Istio ingress gateway with this command:

```
$ kubectl create ns foo
$ kubectl apply -f <(istioctl kube-inject -f @samples/httpbin/httpbin.yaml@) -n foo
$ kubectl apply -f <(istioctl kube-inject -f @samples/httpbin-gateway.yaml@) -n foo
```

- Turn on RBAC debugging in Envoy for the ingress

## gateway:

```
$ kubectl get pods -n istio-system -o name -l istio=ingress gateway | sed 's|pod/||' | while read -r pod; do istioctl proxy-config log "$pod" -n istio-system --level rbac:debug; done
```

- Follow the instructions in Determining the ingress IP and ports to define the `INGRESS_HOST` and `INGRESS_PORT` environment variables.
- Verify that the `httpbin` workload and ingress gateway are working as expected using this command:

```
$ curl "$INGRESS_HOST:$INGRESS_PORT"/headers -s -o /dev/null  
1 -w "%{http_code}\n"  
200
```



If you don't see the expected output, retry after a few seconds. Caching and propagation overhead can cause a delay.

# Getting traffic into Kubernetes and Istio

All methods of getting traffic into Kubernetes involve opening a port on all worker nodes. The main features that accomplish this are the `NodePort` service and the `LoadBalancer` service. Even the Kubernetes Ingress resource must be backed by an Ingress controller that will create either a `NodePort` or a `LoadBalancer` service.

- A `NodePort` just opens up a port in the range 30000-32767 on each worker node and uses a label selector to identify which Pods to send the traffic to. You have to manually create some kind of load balancer in front of your worker nodes or use Round-Robin DNS.

- A LoadBalancer is just like a NodePort, except it also creates an environment specific external load balancer to handle distributing traffic to the worker nodes. For example, in AWS EKS, the LoadBalancer service will create a Classic ELB with your worker nodes as targets. If your Kubernetes environment does not have a LoadBalancer implementation, then it will just behave like a NodePort. An Istio ingress gateway creates a LoadBalancer service.

What if the Pod that is handling traffic from the NodePort or LoadBalancer isn't running on the worker node that received the traffic? Kubernetes has its own

internal proxy called kube-proxy that receives the packets and forwards them to the correct node.

## **Source IP address of the original client**

If a packet goes through an external proxy load balancer and/or kube-proxy, then the original source IP address of the client is lost. Below are some strategies for preserving the original client IP for logging or security purposes.

# TCP/UDP Proxy Load Balancer

Network Load Balancer

HTTP/HTTPS Load Balancer

A critical bug has been identified in Envoy that the proxy protocol downstream address is restored incorrectly for non-HTTP connections.

Please DO NOT USE the `remoteIpBlocks` field and `remote_ip` attribute with proxy protocol on non-HTTP connections



until a newer version of Istio is released with a proper fix.

Note that Istio doesn't support the proxy protocol and it can be enabled only with the EnvoyFilter API and should be used at your own risk.

If you are using a TCP/UDP Proxy external load balancer (AWS Classic ELB), it can use the Proxy Protocol to embed the original client IP address in the packet data. Both the external load balancer and the Istio ingress gateway must

support the proxy protocol for it to work. In Istio, you can enable it with an EnvoyFilter like below:

```
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: proxy-protocol
  namespace: istio-system
spec:
  configPatches:
    - applyTo: LISTENER
      patch:
        operation: MERGE
        value:
          listener_filters:
            - name: envoy.listener.proxy_protocol
            - name: envoy.listener.tls_inspector
  workloadSelector:
    labels:
      istio: ingressgateway
```

Here is a sample of the IstioOperator that shows

# how to configure the Istio ingress gateway on AWS EKS to support the Proxy Protocol:

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  meshConfig:
    accessLogEncoding: JSON
    accessLogFile: /dev/stdout
  components:
    ingressGateways:
      - enabled: true
        k8s:
          hpaSpec:
            maxReplicas: 10
            minReplicas: 5
          serviceAnnotations:
            service.beta.kubernetes.io/aws-load-balancer-access-log-emit-interval: "5"
```

```
        service.beta.kubernetes.io/aws-load-balancer-access-log-enabled: "true"
        service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-name: elb-logs
        service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-prefix: k8sELBIngressGW
        service.beta.kubernetes.io/aws-load-balancer-proxy-protocol: "*"
    affinity:
        podAntiAffinity:
            preferredDuringSchedulingIgnoredDuringExecution:
                - podAffinityTerm:
                    labelSelector:
                        matchLabels:
                            istio: ingressgateway
                            topologyKey: failure-domain.beta.kubernetes.io/zone
                        weight: 1
                name: istio-ingressgateway
```

For reference, here are the types of load balancers created by Istio with a `LoadBalancer` service on popular managed Kubernetes environments:

Cloud Provider	Load Balancer Name	Load Balance
AWS EKS	Classic Elastic Load Balancer	TCP Proxy
GCP GKE	TCP/UDP Network Load Balancer	Network

Azure AKS	Azure Load Balancer	Network
DO DOKS	Load Balancer	Network

You can instruct AWS EKS to create a Network Load Balancer when you install Istio by using a `serviceAnnotation` like below:

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  meshConfig:
    accessLogEncoding: JSON
    accessLogFile: /dev/stdout
  components:
    ingressGateways:
      - enabled: true
    k8s:
      hpaSpec:
        maxReplicas: 10
        minReplicas: 5
      serviceAnnotations:
        service.beta.kubernetes.io/aws-load-balancer
        r-type: "nlb"
```

# IP-based allow list and deny list

**When to use ipBlocks vs. remoteIpBlocks:** If you are using the X-Forwarded-For HTTP header or the Proxy Protocol to determine the original client IP address, then you should use `remoteIpBlocks` in your `AuthorizationPolicy`. If you are using `externalTrafficPolicy: Local`, then you should use `ipBlocks` in your `AuthorizationPolicy`.

Load Balancer Type

Source of Client IP

ipBlocks v

TCP Proxy	Proxy Protocol	remoteIpBlock
Network	packet source address	ipBlocks
HTTP/HTTPS	X-Forwarded-For	remoteIpBlock

- The following command creates the authorization policy, `ingress-policy`, for the Istio ingress gateway. The following policy sets the `action` field to `ALLOW` to allow the IP addresses specified in the `ipBlocks` to access the ingress gateway. IP addresses not in the list will be denied. The

`ipBlocks` supports both single IP address and CIDR notation.

**ipBlocks**

remoteIpBlocks

Create the AuthorizationPolicy:

```
$ kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: ingress-policy
  namespace: istio-system
spec:
  selector:
    matchLabels:
      app: istio-ingressgateway
  action: ALLOW
  rules:
  - from:
    - source:
        ipBlocks: ["1.2.3.4", "5.6.7.0/24"]
EOF
```

- Verify that a request to the ingress gateway is denied:

```
$ curl "$INGRESS_HOST:$INGRESS_PORT"/headers -s -o /dev/null  
1 -w "%{http_code}\n"  
403
```

- Update the `ingress-policy` to include your client IP address:

**ipBlocks**

remoteIpBlocks

Find your original client IP address if you don't know it and assign it to a variable:

```
$ CLIENT_IP=$(kubectl get pods -n istio-system -o name -l
istio=ingressgateway | sed 's|pod/||' | while read -r po
d; do kubectl logs "$pod" -n istio-system | grep remoteIP
; done | tail -1 | awk -F, '{print $3}' | awk -F: '{print
$2}' | sed 's/ //') && echo "$CLIENT_IP"
192.168.10.15
```

```
$ kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: ingress-policy
  namespace: istio-system
spec:
  selector:
    matchLabels:
      app: istio-ingressgateway
  action: ALLOW
  rules:
  - from:
    - source:
        ipBlocks: ["1.2.3.4", "5.6.7.0/24", "$CLIENT_IP"]
EOF
```

- Verify that a request to the ingress gateway is allowed:

```
$ curl "$INGRESS_HOST:$INGRESS_PORT"/headers -s -o /dev/null  
1 -w "%{http_code}\n"  
200
```

- Update the `ingress-policy` authorization policy to set the `action` key to `DENY` so that the IP addresses specified in the `ipBlocks` are not allowed to access the ingress gateway:

**ipBlocks**

remoteIpBlocks

```
$ kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: ingress-policy
  namespace: istio-system
spec:
  selector:
    matchLabels:
      app: istio-ingressgateway
  action: DENY
  rules:
  - from:
    - source:
        ipBlocks: ["$CLIENT_IP"]
EOF
```

- Verify that a request to the ingress gateway is denied:

```
$ curl "$INGRESS_HOST:$INGRESS_PORT"/headers -s -o /dev/null  
1 -w "%{http_code}\n"  
403
```

- You could use an online proxy service to access the ingress gateway using a different client IP to verify the request is allowed.
- If you are not getting the responses you expect, view the ingress gateway logs which should show RBAC debugging information:

```
$ kubectl get pods -n istio-system -o name -l istio=ingress  
gateway | sed 's|pod/||' | while read -r pod; do kubectl lo  
gs "$pod" -n istio-system; done
```

# Clean up

- Remove the namespace foo:

```
$ kubectl delete namespace foo
```

- Remove the authorization policy:

```
$ kubectl delete authorizationpolicy ingress-policy -n isti  
o-system
```

# Trust Domain Migration

⌚ 4 minute read ✓ page test

---

This task shows you how to migrate from one trust domain to another without changing authorization policy.

In Istio 1.4, we introduce an alpha feature to support trust domain migration for authorization policy. This

means if an Istio mesh needs to change its trust domain, the authorization policy doesn't need to be changed manually. In Istio, if a workload is running in namespace `foo` with the service account `bar`, and the trust domain of the system is `my-td`, the identity of said workload is `spiffe://my-td/ns/foo/sa/bar`. By default, the Istio mesh trust domain is `cluster.local`, unless you specify it during the installation.

## Before you begin

Before you begin this task, do the following:

1. Read the Istio authorization concepts.
2. Install Istio with a custom trust domain and mutual TLS enabled.

```
$ istioctl install --set profile=demo --set meshConfig.trus  
tDomain=old-td
```

3. Deploy the httpbin sample in the default namespace and the sleep sample in the default and sleep-allow namespaces:

```
$ kubectl label namespace default istio-injection=enabled
$ kubectl apply -f @samples/httpbin/httpbin.yaml@
$ kubectl apply -f @samples/sleep/sleep.yaml@
$ kubectl create namespace sleep-allow
$ kubectl label namespace sleep-allow istio-injection=enabled
$ kubectl apply -f @samples/sleep/sleep.yaml@ -n sleep-allow
```

4. Apply the authorization policy below to deny all requests to httpbin except from sleep in the sleep-allow namespace.

```
$ kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: service-httpbin.default.svc.cluster.local
```

```
namespace: default
spec:
  rules:
    - from:
      - source:
          principals:
            - old-td/ns/sleep-allow/sa/sleep
      to:
        - operation:
            methods:
              - GET
      selector:
        matchLabels:
          app: httpbin
...
EOF
```

Notice that it may take tens of seconds for the authorization policy to be propagated to the sidecars.

# 1. Verify that requests to httpbin from:

- sleep in the default namespace are denied.

```
$ kubectl exec "$(kubectl get pod -l app=sleep -o jsonpath={.items..metadata.name})" -c sleep -- curl http://httpbin.default:8000/ip -ss -o /dev/null -w "%{http_code}\n"  
403
```

- sleep in the sleep-allow namespace are allowed.

```
$ kubectl exec "$(kubectl -n sleep-allow get pod -l app=sleep -o jsonpath={.items..metadata.name})" -c sleep -n sleep-allow -- curl http://httpbin.default:8000/ip -ss -o /dev/null -w "%{http_code}\n"  
200
```

# Migrate trust domain without trust domain aliases

1. Install Istio with a new trust domain.

```
$ istioctl install --set profile=demo --set meshConfig.trus  
tDomain=new-td
```

2. Redeploy istiod to pick up the trust domain changes.

```
$ kubectl rollout restart deployment -n istio-system istiod
```

Istio mesh is now running with a new trust domain, new-td.

3. Redeploy the httpbin and sleep applications to pick up changes from the new Istio control plane.

```
$ kubectl delete pod --all
```

```
$ kubectl delete pod --all -n sleep-allow
```

4. Verify that requests to httpbin from both sleep in default namespace and sleep-allow namespace are denied.

```
$ kubectl exec "$(kubectl get pod -l app=sleep -o jsonpath={.items..metadata.name})" -c sleep -- curl http://httpbin.default:8000/ip -ss -o /dev/null -w "%{http_code}\n"  
403
```

```
$ kubectl exec "$(kubectl -n sleep-allow get pod -l app=sleep -o jsonpath={.items..metadata.name})" -c sleep -n sleep-allow -- curl http://httpbin.default:8000/ip -ss -o /dev/null -w "%{http_code}\n"  
403
```

This is because we specified an authorization policy that deny all requests to `httpbin`, except the ones the `old-td/ns/sleep-allow/sa/sleep` identity, which is the old identity of the `sleep` application in `sleep-allow` namespace. When we migrated to a

new trust domain above, i.e. new-td, the identity of this sleep application is now new-td/ns/sleep-allow/sa/sleep, which is not the same as old-td/ns/sleep-allow/sa/sleep. Therefore, requests from the sleep application in sleep-allow namespace to httpbin were allowed before are now being denied. Prior to Istio 1.4, the only way to make this work is to change the authorization policy manually. In Istio 1.4, we introduce an easy way, as shown below.

# **Migrate trust domain with trust domain aliases**

1. Install Istio with a new trust domain and trust domain aliases.

```
$ cat <<EOF > ./td-installation.yaml
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  meshConfig:
    trustDomain: new-td
    trustDomainAliases:
      - old-td
EOF
$ istioctl install --set profile=demo -f td-installation.yaml -y
```

2. Without changing the authorization policy, verify that requests to `httpbin` from:
  - `sleep` in the default namespace are denied.

```
$ kubectl exec "$(kubectl get pod -l app=sleep -o jsonpath={.items..metadata.name})" -c sleep -- curl http://httpbin.default:8000/ip -ss -o /dev/null -w "%{http_code}\n"
403
```

- sleep in the sleep-allow namespace are allowed.

```
$ kubectl exec "$(kubectl -n sleep-allow get pod -l app=sleep -o jsonpath={.items..metadata.name})" -c sleep -n sleep-allow -- curl http://httpbin.default:8000/ip -ss -o /dev/null -w "%{http_code}\n"
200
```

# Best practices

Starting from Istio 1.4, when writing authorization policy, you should consider using the value `cluster.local` as the trust domain part in the policy. For example, instead of `old-td/ns/sleep-allow/sa/sleep`, it should be `cluster.local/ns/sleep-allow/sa/sleep`. Notice that in this case, `cluster.local` is not the Istio mesh trust domain (the trust domain is still `old-td`). However, in authorization policy, `cluster.local` is a pointer that points to the current trust domain, i.e. `old-td` (and later `new-td`), as well as its aliases. By using `cluster.local` in the authorization policy, when

you migrate to a new trust domain, Istio will detect this and treat the new trust domain as the old trust domain without you having to include the aliases.

## Clean up

```
$ kubectl delete authorizationpolicy service-httpbin.default.svc  
.cluster.local  
$ kubectl delete deploy httpbin; kubectl delete service httpbin;  
kubectl delete serviceaccount httpbin  
$ kubectl delete deploy sleep; kubectl delete service sleep; kub  
ectl delete serviceaccount sleep  
$ istioctl x uninstall --purge  
$ kubectl delete namespace sleep-allow istio-system  
$ rm ./td-installation.yaml
```

# Dry Run

⌚ 4 minute read  page test

---

This task shows you how to set up an Istio authorization policy using a new experimental annotation `istio.io/dry-run` to dry-run the policy without actually enforcing it.

The dry-run annotation allows you to better understand the effect of an authorization policy

before applying it to the production traffic. This helps to reduce the risk of breaking the production traffic caused by an incorrect authorization policy.



The following information describes an experimental feature, which is intended for evaluation purposes only.

## Before you begin

Before you begin this task, do the following:

- Read the Istio authorization concepts.
- Follow the Istio installation guide **to install Istio**.
- Deploy Zipkin for checking dry-run tracing results. Follow the Zipkin task to install Zipkin in the cluster. Make sure the sampling rate is set to 100 which allows you to quickly reproduce the trace span in the task.
- Deploy Prometheus for checking dry-run metric results. Follow the Prometheus task **to install the Prometheus** in the cluster.

- Deploy test workloads:

This task uses two workloads, httpbin and sleep, both deployed in namespace foo. Both workloads run with an Envoy proxy sidecar. Create the foo namespace and deploy the workloads with the following command:

```
$ kubectl create ns foo
$ kubectl label ns foo istio-injection=enabled
$ kubectl apply -f @samples/httpbin/httpbin.yaml@ -n foo
$ kubectl apply -f @samples/sleep/sleep.yaml@ -n foo
```

- Enable proxy debug level log for checking dry-run logging results:

```
$ istioctl proxy-config log deploy/httpbin.foo --level "rbac:debug" | grep rbac
rbac: debug
```

- Verify that sleep can access httpbin with the following command:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})" -c sleep -n foo -- curl http://httpbin.foo:8000/ip -s -o /dev/null -w "%{http_code}\n"
200
```

If you don't see the expected output as you follow the task, retry after a few seconds.

Caching and propagation overhead can cause some delay.

## Create dry-run policy

1. Create an authorization policy with dry-run annotation "istio.io/dry-run": "true" with the following command:

```
$ kubectl apply -n foo -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: deny-path-headers
  annotations:
    "istio.io/dry-run": "true"
spec:
  selector:
    matchLabels:
      app: httpbin
  action: DENY
  rules:
  - to:
    - operation:
        paths: ["/headers"]
EOF
```

2. Verify a request to path /headers is allowed

because the policy is created in dry-run mode:

```
$ kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})" -c sleep -n foo -- curl "http://httpbin.foo:8000/headers" -s
```

## Check dry-run result in proxy log

1. The dry-run results can be found in the proxy debug log, similar to shadow denied, matched policy ns[foo]-policy[deny-path-headers]-rule[0]. See the

troubleshooting guide for more details.

## Check dry-run result in metric using Prometheus

1. Open the Prometheus dashboard with the following command:

```
$ istioctl dashboard prometheus
```

2. In the Prometheus dashboard, search for the

## metric

envoy\_http\_inbound\_0\_0\_0\_0\_80\_rbac{authz\_dry\_run\_result!=""}}. The following is an example metric output:

```
envoy_http_inbound_0_0_0_0_80_rbac{app="httpbin", authz_dry_
run_action="deny", authz_dry_run_result="allowed", instance="
10.44.1.11:15020", istio_io_rev="default", job="kubernetes-po
ds", kubernetes_namespace="foo", kubernetes_pod_name="httpbin
-74fb669cc6-95qm8", pod_template_hash="74fb669cc6", security_
istio_io_tlsMode="istio", service_istio_io_canonical_name="h
ttpbin", service_istio_io_canonical_revision="v1", version="v
1"} 0
envoy_http_inbound_0_0_0_0_80_rbac{app="httpbin", authz_dry_
run_action="deny", authz_dry_run_result="denied", instance="1
0.44.1.11:15020", istio_io_rev="default", job="kubernetes-pod
s", kubernetes_namespace="foo", kubernetes_pod_name="httpbin-
74fb669cc6-95qm8", pod_template_hash="74fb669cc6", security_i
stio_io_tlsMode="istio", service_istio_io_canonical_name="ht
tpbin", service_istio_io_canonical_revision="v1", version="v1
"} 1
```

### 3. The metric

```
envoy_http_inbound_0_0_0_0_80_rbac{authz_dry_run_res
```

`ult="denied"}` has value `1` (you might find different value depending on how many requests you have sent. It's expected as long as the value is greater than `0`). This means the dry-run policy applied to the `httpbin` workload on port `80` matched one request. The policy would reject the request once if it was not in dry-run mode.

## Check dry-run result in tracing using Zipkin

1. Open the Zipkin dashboard with the following command:

```
$ istioctl dashboard zipkin
```

2. Find the trace result for the request from `sleep` to `httpbin`. Try to send some more requests if you do see the trace result due to the delay in the Zipkin.
3. In the trace result, you should find the following custom tags indicating the request is rejected by the dry-run policy `deny-path-headers` in the namespace `foo`:

```
istio.authorization.dry_run.deny_policy.name: ns[foo]-policy[deny-path-headers]-rule[0]
istio.authorization.dry_run.deny_policy.result: denied
```

## Summary

The Proxy debug log, Prometheus metric and Zipkin trace results indicate that the dry-run policy will reject the request. You can further change the policy if the dry-run result is not expected.

It's recommended to keep the dry-run policy for some

additional time so that it can be tested with more production traffic.

When you are confident about the dry-run result, you can disable the dry-run mode so that the policy will start to actually reject requests. This can be achieved by either of the following approaches:

- Remove the dry-run annotation completely; or
- Change the value of the dry-run annotation to false.

# Limitations

The dry-run annotation is currently in experimental stage and has the following limitations:

- The dry-run annotation currently only supports ALLOW and DENY policies;
- There will be two separate dry-run results (i.e. log, metric and tracing tag) for ALLOW and DENY policies due to the fact that the ALLOW and DENY policies are enforced separately in the proxy. You should take all the two dry-run results into consideration because a request could be

allowed by an ALLOW policy but still rejected by another DENY policy;

- The dry-run results in the proxy log, metric and tracing are for manual troubleshooting purposes and should not be used as an API because it may change anytime without prior notice.

## Clean up

1. Remove the namespace `foo` from your configuration:

```
$ kubectl delete namespace foo
```

2. Remove Prometheus and Zipkin if no longer needed.

# Enabling Rate Limits using Envoy

⌚ 6 minute read  page test

---

This task shows you how to use Envoy's native rate limiting to dynamically limit the traffic to an Istio service. In this task, you will apply a global rate-limit for the `productpage` service through ingress gateway that allows 1 requests per minute across all instances

of the service. Additionally, you will apply a local rate-limit for each individual productpage instance that will allow 10 requests per minute. In this way, you will ensure that the productpage service handles a maximum of 1 request per minute through the ingress gateway, but each productpage instance can handle up to 10 requests per minute, allowing for any in-mesh traffic.

## Before you begin

1. Setup Istio in a Kubernetes cluster by following the instructions in the Installation Guide.
2. Deploy the Bookinfo sample application.

## Rate limits

Envoy supports two kinds of rate limiting: global and local. Global rate limiting uses a global gRPC rate limiting service to provide rate limiting for the entire mesh. Local rate limiting is used to limit the rate of requests per service instance. Local rate limiting can

be used in conjunction with global rate limiting to reduce load on the global rate limiting service.

In this task you will configure Envoy to rate limit traffic to a specific path of a service using both global and local rate limits.

## Global rate limit

Envoy can be used to set up global rate limits for your mesh. Global rate limiting in Envoy uses a gRPC API for requesting quota from a rate limiting service. A

reference implementation of the API, written in Go with a Redis backend, is used below.

1. Use the following configmap to configure the reference implementation to rate limit requests to the path `/productpage` at 1 req/min and all other requests at 100 req/min.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: ratelimit-config
data:
  config.yaml: |
    domain: productpage-ratelimit
    descriptors:
      - key: PATH
        value: "/productpage"
        rate_limit:
          unit: minute
          requests_per_unit: 1
      - key: PATH
        rate_limit:
          unit: minute
          requests_per_unit: 100
```

2. Create a global rate limit service which

implements Envoy's rate limit service protocol. As a reference, a demo configuration can be found here, which is based on a reference implementation provided by Envoy.

3. Apply an `EnvoyFilter` to the `ingressgateway` to enable global rate limiting using Envoy's global rate limit filter.

The first patch inserts the `envoy.filters.http.ratelimit` global envoy filter **filter** into the `HTTP_FILTER` chain. The `rate_limit_service` field specifies the external rate limit service, `rate_limit_cluster` in this case.

The second patch defines the `rate_limit_cluster`,

which provides the endpoint location of the external rate limit service.

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: filter-ratelimit
  namespace: istio-system
spec:
  workloadSelector:
    # select by label in the same namespace
    labels:
      istio: ingressgateway
  configPatches:
    # The Envoy config you want to modify
    - applyTo: HTTP_FILTER
      match:
        context: GATEWAY
```

```
    listener:  
      filterChain:  
        filter:  
          name: "envoy.filters.network.http_connection_manager"  
            subFilter:  
              name: "envoy.filters.http.router"  
        patch:  
          operation: INSERT_BEFORE  
          # Adds the Envoy Rate Limit Filter in HTTP filter chain.  
          value:  
            name: envoy.filters.http.ratelimit  
            typed_config:  
              "@type": type.googleapis.com/envoy.extensions.filters.http.ratelimit.v3.RateLimit  
                # domain can be anything! Match it to the rate limiter service config  
                domain: productpage-ratelimit  
                failure_mode_deny: true
```

```
    timeout: 10s
    rate_limit_service:
        grpc_service:
            envoy_grpc:
                cluster_name: rate_limit_cluster
                transport_api_version: V3
- applyTo: CLUSTER
match:
    cluster:
        service: ratelimit.default.svc.cluster.local
patch:
    operation: ADD
    # Adds the rate limit service cluster for rate limit service defined in step 1.
    value:
        name: rate_limit_cluster
        type: STRICT_DNS
        connect_timeout: 10s
        lb_policy: ROUND_ROBIN
        http2_protocol_options: {}
```

```
load_assignment:  
    cluster_name: rate_limit_cluster  
    endpoints:  
        - lb_endpoints:  
            - endpoint:  
                address:  
                    socket_address:  
                        address: ratelimit.default.svc.cluster.local  
                        port_value: 8081  
EOF
```

4. Apply another EnvoyFilter to the ingressgateway that defines the route configuration on which to rate limit. This adds rate limit actions for any route from a virtual host named \*.80.

```
$ kubectl apply -f - <<EOF
```

```
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: filter-ratelimit-svc
  namespace: istio-system
spec:
  workloadSelector:
    labels:
      istio: ingressgateway
  configPatches:
    - applyTo: VIRTUAL_HOST
      match:
        context: GATEWAY
        routeConfiguration:
          vhost:
            name: ""
            route:
              action: ANY
      patch:
        operation: MERGE
```

```
# Applies the rate limit rules.  
value:  
    rate_limits:  
        - actions: # any actions in here  
        - request_headers:  
            header_name: ":path"  
            descriptor_key: "PATH"  
EOF
```

## Local rate limit

Envoy supports local rate limiting of L4 connections and HTTP requests. This allows you to apply rate limits at the instance level, in the proxy itself, without calling

any other service.

The following EnvoyFilter enables local rate limiting for any traffic through the productpage service. The HTTP\_FILTER patch inserts the envoy.filters.http.local\_ratelimit local envoy filter into the HTTP connection manager filter chain. The local rate limit filter's token bucket is configured to allow 10 requests/min. The filter is also configured to add an x-local-rate-limit response header to requests that are blocked.

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
```

```
kind: EnvoyFilter
metadata:
  name: filter-local-ratelimit-svc
  namespace: istio-system
spec:
  workloadSelector:
    labels:
      app: productpage
  configPatches:
    - applyTo: HTTP_FILTER
      match:
        context: SIDECAR_INBOUND
        listener:
          filterChain:
            filter:
              name: "envoy.filters.network.http_connection_manager"
      patch:
        operation: INSERT_BEFORE
        value:
```

```
name: envoy.filters.http.local_ratelimit
typed_config:
  "@type": type.googleapis.com/udpa.type.v1.TypedStruct
  type_url: type.googleapis.com/envoy.extensions.filters.http.local_ratelimit.v3.LocalRateLimit
  value:
    stat_prefix: http_local_rate_limiter
    token_bucket:
      max_tokens: 10
      tokens_per_fill: 10
      fill_interval: 60s
    filter_enabled:
      runtime_key: local_rate_limit_enabled
    default_value:
      numerator: 100
      denominator: HUNDRED
    filter_enforced:
      runtime_key: local_rate_limit_enforced
    default_value:
```

```
    numerator: 100
    denominator: HUNDRED
  response_headers_to_add:
    - append: false
      header:
        key: x-local-rate-limit
        value: 'true'
```

EOF

The above configuration applies local rate limiting to all vhosts/routes. Alternatively, you can restrict it to a specific route.

The following EnvoyFilter enables local rate limiting for any traffic to port 80 of the productpage service. Unlike the previous configuration, there is no

token\_bucket included in the HTTP\_FILTER patch. The token\_bucket is instead defined in the second (HTTP\_ROUTE) patch which includes a typed\_per\_filter\_config for the envoy.filters.http.local\_ratelimit local envoy filter, for routes to virtual host inbound|http|9080.

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: filter-local-ratelimit-svc
  namespace: istio-system
spec:
  workloadSelector:
    labels:
      app: productpage
```

```
configPatches:  
  - applyTo: HTTP_FILTER  
    match:  
      context: SIDECAR_INBOUND  
      listener:  
        filterChain:  
          filter:  
            name: "envoy.filters.network.http_connection_manager"  
er"  
  patch:  
    operation: INSERT_BEFORE  
    value:  
      name: envoy.filters.http.local_ratelimit  
      typed_config:  
        "@type": type.googleapis.com/udpa.type.v1.TypedStruct  
        type_url: type.googleapis.com/envoy.extensions.filters.http.local_ratelimit.v3.LocalRateLimit  
        value:  
          stat_prefix: http_local_rate_limiter
```

```
- applyTo: HTTP_ROUTE
  match:
    context: SIDECAR_INBOUND
    routeConfiguration:
      vhost:
        name: "inbound|http|9080"
        route:
          action: ANY
  patch:
    operation: MERGE
    value:
      typed_per_filter_config:
        envoy.filters.http.local_ratelimit:
          "@type": type.googleapis.com/udpa.type.v1.TypedStruct
          type_url: type.googleapis.com/envoy.extensions.filters.http.local_ratelimit.v3.LocalRateLimit
          value:
            stat_prefix: http_local_rate_limiter
            token_bucket:
```

```
    max_tokens: 10
    tokens_per_fill: 10
    fill_interval: 60s
filter_enabled:
    runtime_key: local_rate_limit_enabled
default_value:
    numerator: 100
    denominator: HUNDRED
filter_enforced:
    runtime_key: local_rate_limit_enforced
default_value:
    numerator: 100
    denominator: HUNDRED
response_headers_to_add:
    - append: false
      header:
          key: x-local-rate-limit
          value: 'true'
```

EOF

# Verify the results

## Verify global rate limit

Send traffic to the Bookinfo sample. Visit

`http://$GATEWAY_URL/productpage` in your web browser or issue the following command:

```
$ curl "http://$GATEWAY_URL/productpage"
```

`$GATEWAY_URL` is the value set in the Bookinfo

example.

You will see the first request go through but every following request within a minute will get a 429 response.

## Verify local rate limit

Although the global rate limit at the ingress gateway limits requests to the productpage service at 1 req/min, the local rate limit for productpage instances allows 10

req/min. To confirm this, send internal productpage requests, from the ratings pod, using the following curl command:

```
$ kubectl exec "$(kubectl get pod -l app=ratings -o jsonpath='{.items[0].metadata.name}')" -c ratings -- curl -sS productpage:9080/productpage | grep -o "<title>.*</title>"  
<title>Simple Bookstore App</title>
```

You should see no more than 10 req/min go through per productpage instance.

# Collecting Metrics for TCP Services

⌚ 4 minute read ✓ page test

---

This task shows how to configure Istio to automatically gather telemetry for TCP services in a mesh. At the end of this task, you can query default TCP metrics for your mesh.

The Bookinfo sample application is used as the

example throughout this task.

# Before you begin

- Install Istio in your cluster and deploy an application. You must also install Prometheus.
- This task assumes that the Bookinfo sample will be deployed in the `default` namespace. If you use a different namespace, update the example configuration and commands.

# Collecting new telemetry data

1. Setup Bookinfo to use MongoDB.
  1. Install v2 of the ratings service.

If you are using a cluster with automatic sidecar injection enabled, deploy the services using `kubectl`:

```
$ kubectl apply -f @samples/bookinfo/platform/kube/bookinfo-ratings-v2.yaml@  
serviceaccount/bookinfo-ratings-v2 created  
deployment.apps/ratings-v2 created
```

If you are using manual sidecar injection, run the following command instead:

```
$ kubectl apply -f <(istioctl kube-inject -f @samples/bookinfo/platform/kube/bookinfo-ratings-v2.yaml@)
deployment "ratings-v2" configured
```

## 2. Install the `mongodb` service:

If you are using a cluster with automatic sidecar injection enabled, deploy the services using `kubectl`:

```
$ kubectl apply -f @samples/bookinfo/platform/kube/bookinfo-db.yaml@
service/mongodb created
deployment.apps/mongodb-v1 created
```

If you are using manual sidecar injection, run the following command instead:

```
$ kubectl apply -f <(istioctl kube-inject -f @samples/bookinfo/platform/kube/bookinfo-db.yaml@)
service "mongodb" configured
deployment "mongodb-v1" configured
```

3. The Bookinfo sample deploys multiple versions of each microservice, so begin by creating destination rules that define the service subsets corresponding to each version, and the load balancing policy for each subset.

```
$ kubectl apply -f @samples/bookinfo/networking/destination-rule-all.yaml@
```

If you enabled mutual TLS, run the following command instead:

```
$ kubectl apply -f @samples/bookinfo/networking/destination-rule-all-mtls.yaml@
```

To display the destination rules, run the following command:

```
$ kubectl get destinationrules -o yaml
```

Wait a few seconds for destination rules to propagate before adding virtual services that

refer to these subsets, because the subset references in virtual services rely on the destination rules.

#### 4. Create ratings and reviews virtual services:

```
$ kubectl apply -f @samples/bookinfo/networking/virtual  
-service-ratings-db.yaml@  
virtualservice.networking.istio.io/reviews created  
virtualservice.networking.istio.io/ratings created
```

#### 2. Send traffic to the sample application.

For the Bookinfo sample, visit

`http://$GATEWAY_URL/productpage` in your web browser or use the following command:

```
$ curl http://"${GATEWAY_URL}/productpage"
```



`$GATEWAY_URL` is the value set in the Bookinfo example.

3. Verify that the TCP metric values are being generated and collected.

In a Kubernetes environment, setup port-forwarding for Prometheus by using the following command:

```
$ istioctl dashboard prometheus
```

View the values for the TCP metrics in the Prometheus browser window. Select **Graph**. Enter the `istio_tcp_connections_opened_total` metric or `istio_tcp_connections_closed_total` and select **Execute**. The table displayed in the **Console** tab includes entries similar to:

```
istio_tcp_connections_opened_total{  
destination_version="v1",  
instance="172.17.0.18:42422",  
job="istio-mesh",  
canonical_service_name="ratings-v2",  
canonical_service_revision="v2"}
```

```
istio_tcp_connections_closed_total{  
destination_version="v1",  
instance="172.17.0.18:42422",  
job="istio-mesh",  
canonical_service_name="ratings-v2",  
canonical_service_revision="v2"}
```

# Understanding TCP telemetry collection

In this task, you used Istio configuration to automatically generate and report metrics for all

traffic to a TCP service within the mesh. TCP Metrics for all active connections are recorded every 15s by default and this timer is configurable via `tcpReportingDuration`. Metrics for a connection are also recorded at the end of the connection.

## TCP attributes

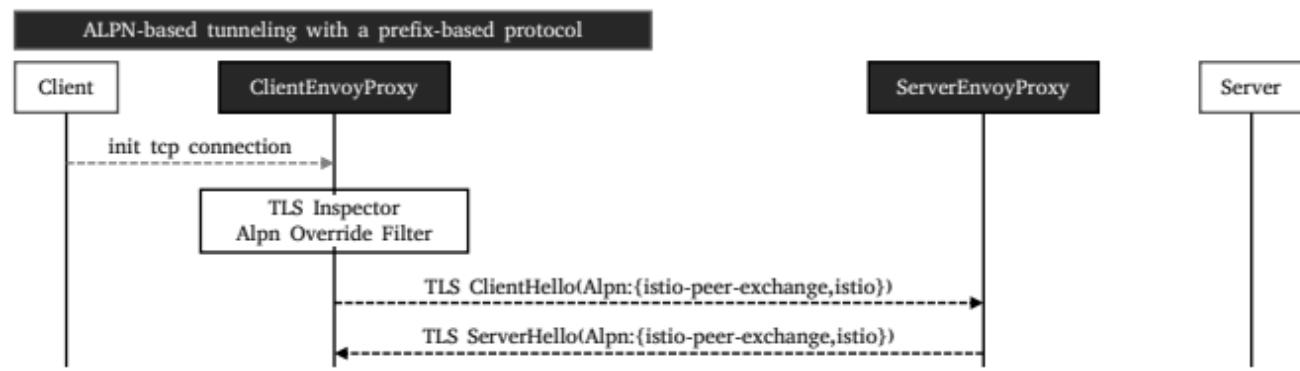
Several TCP-specific attributes enable TCP policy and control within Istio. These attributes are generated by Envoy Proxies and obtained from Istio using Envoy's Node Metadata. Envoy forwards Node Metadata to

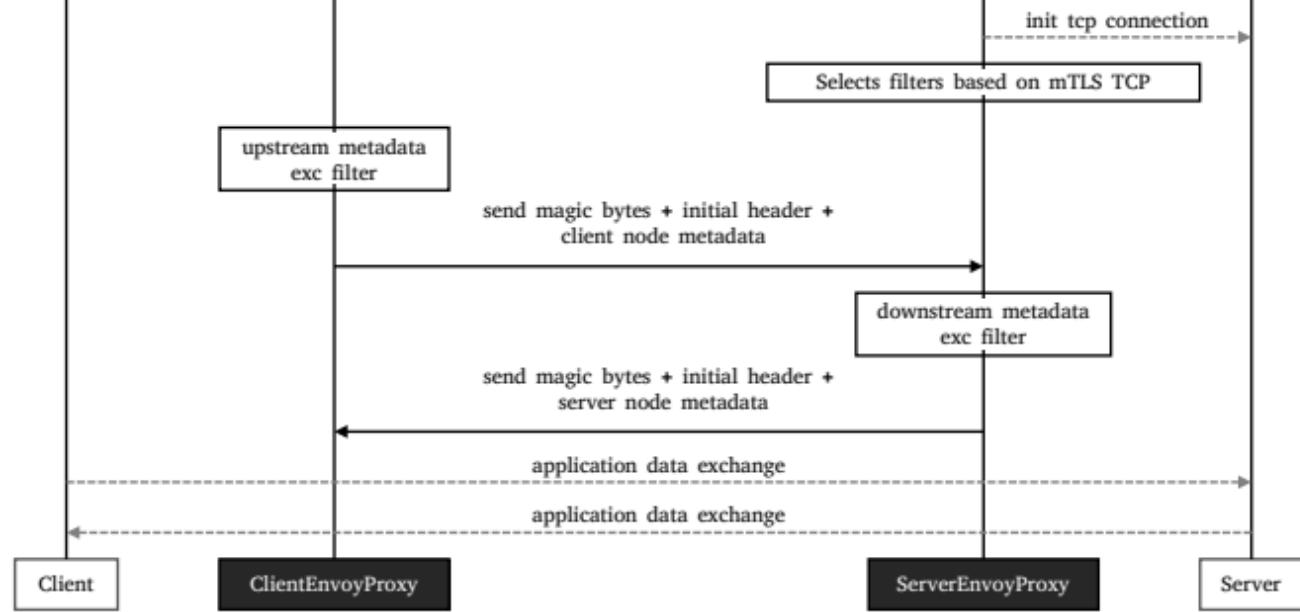
Peer Envoys using ALPN based tunneling and a prefix based protocol. We define a new protocol `istio-peer-exchange`, that is advertised and prioritized by the client and the server sidecars in the mesh. ALPN negotiation resolves the protocol to `istio-peer-exchange` for connections between Istio enabled proxies, but not between an Istio enabled proxy and any other proxy. This protocol extends TCP as follows:

1. TCP client, as a first sequence of bytes, sends a magic byte string and a length prefixed payload.
2. TCP server, as a first sequence of bytes, sends a magic byte sequence and a length prefixed

payload. These payloads are protobuf encoded serialized metadata.

3. Client and server can write simultaneously and out of order. The extension filter in Envoy then does the further processing in downstream and upstream until either the magic byte sequence is not matched or the entire payload is read.





## TCP Attribute Flow

# Cleanup

- Remove the port-forward process:

```
$ killall istioctl
```

- If you are not planning to explore any follow-on tasks, refer to the Bookinfo cleanup instructions to shutdown the application.

# Customizing Istio Metrics

⌚ 4 minute read ✓ page test

---

This task shows you how to customize the metrics that Istio generates.

Istio generates telemetry that various dashboards consume to help you visualize your mesh. For example, dashboards that support Istio include:

- Grafana
- Kiali
- Prometheus

By default, Istio defines and generates a set of standard metrics (e.g. `requests_total`), but you can also customize them and create new metrics.

## Custom statistics configuration

Istio uses the Envoy proxy to generate metrics and provides its configuration in the EnvoyFilter at manifests/charts/istio-control/istio-discovery/templates/telemetryv2\_1.11.yaml.

Configuring custom statistics involves two sections of the EnvoyFilter: definitions and metrics. The definitions section supports creating new metrics by name, the expected value expression, and the metric type (counter, gauge, and histogram). The metrics section provides values for the metric dimensions as expressions, and allows you to remove or override the existing metric dimensions. You can modify the standard metric definitions using tags\_to\_remove or by

re-defining a dimension. These configuration settings are also exposed as `istioctl` installation options, which allow you to customize different metrics for gateways and sidecars as well as for the inbound or outbound direction.

For more information, see [Stats Config reference](#).

## Before you begin

Install Istio in your cluster and deploy an application.

Alternatively, you can set up custom statistics as part of the Istio installation.

The Bookinfo sample application is used as the example application throughout this task.

## Enable custom metrics

1. The default telemetry v2 `EnvoyFilter` configuration is equivalent to the following installation options:

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  values:
    telemetry:
      v2:
        prometheus:
          configOverride:
            inboundSidecar:
              disable_host_headerFallback: false
            outboundSidecar:
              disable_host_headerFallback: false
            gateway:
              disable_host_headerFallback: true
```

To customize telemetry v2 metrics, for example,  
to add `request_host` and `destination_port`  
dimensions to the `requests_total` metric emitted by

both gateways and sidecars in the inbound and outbound direction, change the installation options as follows:

You only need to specify the configuration for the settings that you want to customize. For example, to only customize the sidecar inbound requests\_count metric, you can omit the outboundSidecar and gateway sections in the configuration. Unspecified settings will retain the default configuration, equivalent to the explicit settings shown



above.

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  values:
    telemetry:
      v2:
        prometheus:
          configOverride:
            inboundSidecar:
              metrics:
                - name: requests_total
                  dimensions:
                    destination_port: string(destination.po
rt)
                    request_host: request.host
        outboundSidecar:
```

```
        metrics:
          - name: requests_total
            dimensions:
              destination_port: string(destination.po
rt)
                  request_host: request.host
      gateway:
        metrics:
          - name: requests_total
            dimensions:
              destination_port: string(destination.po
rt)
                  request_host: request.host
```

2. Apply the following annotation to all injected pods with the list of the dimensions to extract into a Prometheus time series using the following command:



This step is needed only if your dimensions are not already in DefaultStatTags list

```
apiVersion: apps/v1
kind: Deployment
spec:
  template: # pod template
  metadata:
    annotations:
      sidecar.istio.io/extrastatTags: destination_port, request_host
```

To enable extra tags mesh wide, you can add

extraStatTags to your mesh config:

```
meshConfig:  
  defaultConfig:  
    extraStatTags:  
      - destination_port  
      - request_host
```

## Verify the results

Send traffic to the mesh. For the Bookinfo sample, visit `http://$GATEWAY_URL/productpage` in your web browser or issue the following command:

```
$ curl "http://$GATEWAY_URL/productpage"
```

i \$GATEWAY\_URL is the value set in the Bookinfo example.

Use the following command to verify that Istio generates the data for your new or modified dimensions:

```
$ kubectl exec "$(kubectl get pod -l app=productpage -o jsonpath =' {.items[0].metadata.name}' )" -c istio-proxy -- curl -sS 'localhost:15000/stats/prometheus' | grep istio_requests_total
```

For example, in the output, locate the metric `istio_requests_total` and verify it contains your new dimension.



It might take a short period of time for the proxies to start applying the config. If the metric is not received, you may retry sending requests after a short wait, and look for the metric again.

# Use expressions for values

The values in the metric configuration are common expressions, which means you must double-quote strings in JSON, e.g. ““string value””. Unlike Mixer expression language, there is no support for the pipe (|) operator, but you can emulate it with the `has` or `in` operator, for example:

```
has(request.host) ? request.host : "unknown"
```

For more information, see [Common Expression Language](#).

Istio exposes all standard Envoy attributes. Peer metadata is available as attributes `upstream_peer` for outbound and `downstream_peer` for inbound with the following fields:

Field	Type	Value
<code>name</code>	<code>string</code>	<b>Name of the pod.</b>
<code>namespace</code>	<code>string</code>	<b>Namespace that the pod runs in.</b>
<code>labels</code>	<code>map</code>	<b>Workload labels.</b>

owner	string	Workload owner.
workload_name	string	Workload name.
platform_metadata	map	Platform metadata with prefixed keys.
istio_version	string	Version identifier for the proxy.
mesh_id	string	Unique identifier for the mesh.
app_cont	list<s>	List of short names for

ainers	tring>	application containers.
cluster_id	string	Identifier for the cluster to which this workload belongs.

For example, the expression for the peer app label to be used in an outbound configuration is

```
upstream_peer.labels['app'].value.
```

For more information, see configuration reference.

# Classifying Metrics Based on Request or Response

⌚ 6 minute read   ✎ page test

---

---

It's useful to visualize telemetry based on the type of requests and responses handled by services in your mesh. For example, a bookseller tracks the number of times book reviews are requested. A book review

request has this structure:

```
GET /reviews/{review_id}
```

Counting the number of review requests must account for the unbounded element `review_id`. GET `/reviews/1` followed by GET `/reviews/2` should count as two requests to get reviews.

Istio lets you create classification rules using the AttributeGen plugin that groups requests into a fixed number of logical operations. For example, you can create an operation named `GetReviews`, which is a common way to identify operations using the Open API

Spec `operationId`. This information is injected into request processing as `istio_operationId` attribute with value equal to `GetReviews`. You can use the attribute as a dimension in Istio standard metrics. Similarly, you can track metrics based on other operations like `ListReviews` and `CreateReviews`.

For more information, see the reference content.

Istio uses the Envoy proxy to generate metrics and provides its configuration in the `EnvoyFilter` at `manifests/charts/istio-control/istio-discovery/templates/telemetryv2_1.11.yaml`. As a result, writing classification rules involves adding attributes

to the EnvoyFilter.

## Classify metrics by request

You can classify requests based on their type, for example `ListReview`, `GetReview`, `CreateReview`.

1. Create a file, for example

`attribute_gen_service.yaml`, and save it with the following contents. This adds the `istio.attributegen` plugin to the EnvoyFilter. It also

creates an attribute, `istio_operationId` and populates it with values for the categories to count as metrics.

This configuration is service-specific since request paths are typically service-specific.

```
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: istio-attribute-gen-filter
spec:
  workloadSelector:
    labels:
      app: reviews
  configPatches:
  - applyTo: HTTP_FILTER
    match:
```

```
context: SIDECAR_INBOUND
proxy:
  proxyVersion: '1\.\.9.*'
listener:
  filterChain:
    filter:
      name: "envoy.http_connection_manager"
      subFilter:
        name: "istio.stats"
patch:
  operation: INSERT_BEFORE
  value:
    name: istio.attributegen
    typed_config:
      "@type": type.googleapis.com/udpa.type.v1.TypedSt
ruct
    type_url: type.googleapis.com/envoy.extensions.fi
lters.http.wasm.v3.Wasm
    value:
      config:
```

```
        configuration:  
            "@type": type.googleapis.com/google.protobuf.StringValue  
            value: |  
            {  
                "attributes": [  
                    {  
                        "output_attribute": "istio_operationId",  
                        "match": [  
                            {  
                                "value": "ListReviews",  
                                "condition": "request.url_path == '/reviews' && request.method == 'GET'"  
                            },  
                            {  
                                "value": "GetReview",  
                                "condition": "request.url_path.matches('^/reviews/[^:alnum:]*$') && request.method == 'GET'"  
                            }  
                        ]  
                    }  
                ]  
            }  
        
```

```
        },
        {
            "value": "CreateReview",
            "condition": "request.url_path
== '/reviews/' && request.method == 'POST'"
        }
    ]
}
]
}
vm_config:
    runtime: envoy.wasm.runtime.null
    code:
        local: { inline_string: "envoy.wasm.attributegen" }
```

2. Apply your changes using the following command:

```
$ kubectl -n istio-system apply -f attribute_gen_service.yaml
```

3. Find the stats-filter-1.11 EnvoyFilter resource from the `istio-system` namespace, using the following command:

```
$ kubectl -n istio-system get envoyfilter | grep ^stats-filter-1.11  
stats-filter-1.11 2d
```

4. Create a local file system copy of the EnvoyFilter configuration, using the following command:

```
$ kubectl -n istio-system get envoyfilter stats-filter-1.11 -o yaml > stats-filter-1.11.yaml
```

5. Open stats-filter-1.11.yaml with a text editor and locate the name: istio.stats extension configuration. Update it to map request\_operation dimension in the requests\_total standard metric to istio\_operationId attribute. The updated configuration file section should look like the following.

```
name: istio.stats
typed_config:
  '@type': type.googleapis.com/udpa.type.v1.TypedStruct
  type_url: type.googleapis.com/envoy.extensions.filters.ht
tp.wasm.v3.Wasm
  value:
    config:
      configuration:
        "@type": type.googleapis.com/google.protobuf.String
Value
  value: |
  {
    "metrics": [
      {
        "name": "requests_total",
        "dimensions": {
          "request_operation": "istio_operationId"
        }
      }
    ]
  }
```

6. Save stats-filter-1.11.yaml and then apply the configuration using the following command:

```
$ kubectl -n istio-system apply -f stats-filter-1.11.yaml
```

7. Add the following configuration to the mesh config. This results in the addition of the request\_operation as a new dimension to the istio\_requests\_total metric. Without it, a new metric with the name

envoy\_request\_operation\_\_somevalue\_\_istio\_requests\_total is created.

```
meshConfig:  
  defaultConfig:  
    extraStatTags:  
      - request_operation
```

8. Generate metrics by sending traffic to your application.
9. After the changes take effect, visit Prometheus and look for the new or changed dimensions, for example `istio_requests_total`.

## Classify metrics by

# response

You can classify responses using a similar process as requests. Do note that the `response_code` dimension already exists by default. The example below will change how it is populated.

1. Create a file, for example

`attribute_gen_service.yaml`, and save it with the following contents. This adds the `istio.attributegen` plugin to the `EnvoyFilter` and generates the `istio_responseClass` attribute used by the stats plugin.

This example classifies various responses, such as grouping all response codes in the 200 range as a 2xx dimension.

```
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: istio-attribute-gen-filter
spec:
  workloadSelector:
    labels:
      app: productpage
  configPatches:
  - applyTo: HTTP_FILTER
    match:
      context: SIDECAR_INBOUND
      proxy:
        proxyVersion: '1\\.9.*'
    listener:
```

```
filterChain:  
    filter:  
        name: "envoy.http_connection_manager"  
        subFilter:  
            name: "istio.stats"  
patch:  
    operation: INSERT_BEFORE  
value:  
    name: istio.attributegen  
    typed_config:  
        "@type": type.googleapis.com/udpa.type.v1.TypedSt  
ruct  
        type_url: type.googleapis.com/envoy.extensions.fi  
lters.http.wasm.v3.Wasm  
        value:  
            config:  
                configuration:  
                    "@type": type.googleapis.com/google.protobuf.  
f.StringValue  
                    value: |
```

```
{  
    "attributes": [  
        {  
            "output_attribute": "istio_response  
Class",  
            "match": [  
                {  
                    "value": "2xx",  
                    "condition": "response.code >=  
200 && response.code <= 299"  
                },  
                {  
                    "value": "3xx",  
                    "condition": "response.code >=  
300 && response.code <= 399"  
                },  
                {  
                    "value": "404",  
                    "condition": "response.code ==  
404"  
                }  
            ]  
        }  
    ]  
}
```

```
        },
        {
            "value": "429",
            "condition": "response.code == 429"
        },
        {
            "value": "503",
            "condition": "response.code == 503"
        },
        {
            "value": "5xx",
            "condition": "response.code >= 500 && response.code <= 599"
        },
        {
            "value": "4xx",
            "condition": "response.code >= 400 && response.code <= 499"
```

```
        }
      ]
    }
  ]
}

vm_config:
  runtime: envoy.wasm.runtime.null
  code:
    local: { inline_string: "envoy.wasm.attribute_gen_service.ya
butegen" }
```

2. Apply your changes using the following command:

```
$ kubectl -n istio-system apply -f attribute_gen_service.yaml
```

3. Find the stats-filter-1.11 EnvoyFilter resource from the `istio-system` namespace, using the

following command:

```
$ kubectl -n istio-system get envoyfilter | grep ^stats-filter-1.11  
stats-filter-1.11 2d
```

4. Create a local file system copy of the EnvoyFilter configuration, using the following command:

```
$ kubectl -n istio-system get envoyfilter stats-filter-1.11  
-o yaml > stats-filter-1.11.yaml
```

5. Open stats-filter-1.11.yaml with a text editor and locate the name: istio.stats extension configuration. Update it to map response\_code dimension in the requests\_total standard metric to

`istio_responseClass` attribute. The updated configuration file section should look like the following.

```
name: istio.stats
typed_config:
  '@type': type.googleapis.com/udpa.type.v1.TypedStruct
  type_url: type.googleapis.com/envoy.extensions.filters.ht
tp.wasm.v3.Wasm
  value:
    config:
      configuration:
        "@type": type.googleapis.com/google.protobuf.String
Value
  value: |
  {
    "metrics": [
      {
        "name": "requests_total",
        "dimensions": {
          "response_code": "istio_responseClass"
        }
      }
    ]
  }
```

6. Save stats-filter-1.11.yaml and then apply the configuration using the following command:

```
$ kubectl -n istio-system apply -f stats-filter-1.11.yaml
```

## Verify the results

1. Generate metrics by sending traffic to your application.
2. Visit Prometheus and look for the new or changed dimensions, for example `2xx`. Alternatively, use

the following command to verify that Istio generates the data for your new dimension:

```
$ kubectl exec pod-name -c istio-proxy -- curl -sS 'localhost:15000/stats/prometheus' | grep istio_
```

In the output, locate the metric (e.g. `istio_requests_total`) and verify the presence of the new or changed dimension.

## Troubleshooting

If classification does not occur as expected, check the following potential causes and resolutions.

Review the Envoy proxy logs for the pod that has the service on which you applied the configuration change. Check that there are no errors reported by the service in the Envoy proxy logs on the pod, (`pod-name`), where you configured classification by using the following command:

```
$ kubectl logs pod-name -c istio-proxy | grep -e "Config Error"  
-e "envoy wasm"
```

Additionally, ensure that there are no Envoy proxy

crashes by looking for signs of restarts in the output of the following command:

```
$ kubectl get pods pod-name
```



# Querying Metrics from Prometheus

⌚ 2 minute read ✓ page test

---

This task shows you how to query for Istio Metrics using Prometheus. As part of this task, you will use the web-based interface for querying metric values.

The Bookinfo sample application is used as the example application throughout this task.

# Before you begin

- Install Istio in your cluster.
- Install the Prometheus Addon.
- Deploy the Bookinfo application.

## Querying Istio metrics

1. Verify that the `prometheus` service is running in your cluster.

In Kubernetes environments, execute the following command:

```
$ kubectl -n istio-system get svc prometheus
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
prometheus      ClusterIP   10.109.160.254 <none>        9090/TCP
4m
```

## 2. Send traffic to the mesh.

For the Bookinfo sample, visit [http://\\$GATEWAY\\_URL/productpage](http://$GATEWAY_URL/productpage) in your web browser or issue the following command:

```
$ curl "http://$GATEWAY_URL/productpage"
```



\$GATEWAY\_URL is the value set in the Bookinfo example.

### 3. Open the Prometheus UI.

In Kubernetes environments, execute the following command:

```
$ istioctl dashboard prometheus
```

Click **Graph** to the right of Prometheus in the header.

### 4. Execute a Prometheus query.

In the “Expression” input box at the top of the web page, enter the text:

```
istio_requests_total
```

Then, click the **Execute** button.

The results will be similar to:

Prometheus   Alerts   Graph   Status ▾   Help

Enable query history

Try experimental React UI

istio\_requests\_total

Load time: 15ms  
Resolution: 14s  
Total time series: 8

**Execute** - insert metric at cursor -

Graph   Console

Moment

Element	Value
istio_requests_total{connection_security_policy="mutual_tls",destination_app="details",destination_canonical_service="details",destination_port="9080",destination_principal="spiffe://cluster.local/ns/default/sa/bookinfo-details",destination_service="details",default_svc=cluster.local,destination_service_name="details",destination_service_namespace="default",destination_version="v1",destination_workload="details-v1",destination_workload_namespace="default",instance="10.12.2.13:15090",job="envoy-stats",namespace="default",pod_name="details-v1-45b6f496d-www5q",reporter="destination",request_protocol="http",response_code="200",response_flags="0"}	8

```
source_app productpage source_canonical_service productpage source_principal spiffe://cluster.local/ns/default/sa/bookinfo-productpage ,source_version="v1",source_workload="productpage-v1",source_namespace="default"
istio_requests_total[connection_security_policy="mTLS",destination_app="productpage",destination_canonical_service="productpage",destination_port="9080",destination_principal="spiffe://cluster.local/ns/default/sa/bookinfo-productpage",destination_service="productpage.default.svc.cluster.local",destination_service_name="productpage",destination_service_namespace="default",destination_workload_namespace="default",instance="10.12.1.15:15090",job="envoy-stats",namespace="default",pod_name="productpage-v1-7d6cf7d7df-rh7t7",reporter="destination",request_protocol="http",response_code="0",response_flags="DC",source_app="istio-ingressgateway",source_canonical_service="istio-ingressgateway",source_principal="spiffe://cluster.local/ns/istio-system/sa/istio-ingressgateway-service-account",source_version="unknown",source_workload="istio-ingressgateway",source_namespace="istio-system")
9
istio_requests_total[connection_security_policy="mTLS",destination_app="productpage",destination_canonical_service="productpage",destination_principal="spiffe://cluster.local/ns/default/sa/bookinfo-productpage",destination_service="productpage.default.svc.cluster.local",destination_service_name="productpage",destination_service_namespace="default",destination_version="v1",destination_workload="productpage-v1",destination_workload_namespace="default",instance="10.12.1.15:15090",job="envoy-stats",namespace="default",pod_name="productpage-v1-7d6cf7d7df-rh7t7",reporter="destination",request_protocol="http",response_code="200",response_flags="DC",source_app="istio-ingressgateway",source_canonical_service="istio-ingressgateway",source_principal="spiffe://cluster.local/ns/istio-system/sa/istio-ingressgateway-service-account",source_version="unknown",source_workload="istio-ingressgateway",source_namespace="istio-system")
```

## Prometheus Query Result

You can also see the query results graphically by selecting the Graph tab underneath the **Execute** button.

The screenshot shows the Prometheus web interface. At the top, there's a navigation bar with links for Prometheus, Alerts, Graph, Status, and Help. Below the navigation bar is a search bar containing the query `istio_requests_total`. To the right of the search bar are three status indicators: "Load time: 8ms", "Resolution: 1s", and "Total time series: 10". Below the search bar is a toolbar with buttons for "Execute" (which is highlighted in blue), "Insert metric at cursor", and "Reset". Underneath the toolbar is a dropdown menu labeled "StageA" with options "Console", "Graph", and "Table". A small preview window shows a single data point from the query results. The main area of the interface is a large, empty white space where the query results would be displayed.



```
#rate(http_requests_insecure_policy{destination_service_name="productpage", destination_service_port="8080", status_code="200"}[1m])
```

Remove Graph

Add Graph

## Prometheus Query Result - Graphical

Other queries to try:

- Total count of all requests to the productpage

service:

```
istio_requests_total{destination_service="productpage.default.svc.cluster.local"}
```

- Total count of all requests to v3 of the reviews service:

```
istio_requests_total{destination_service="reviews.default.svc.cluster.local", destination_version="v3"}
```

This query returns the current total count of all requests to the v3 of the reviews service.

- Rate of requests over the past 5 minutes to all instances of the productpage service:

```
rate(istio_requests_total{destination_service=~"productpage"
.*", response_code="200"})[5m])
```

# About the Prometheus addon

The Prometheus addon is a Prometheus server that comes preconfigured to scrape Istio endpoints to collect metrics. It provides a mechanism for persistent storage and querying of Istio metrics.

For more on querying Prometheus, please read their

querying docs.

# Cleanup

- Remove any `istioctl` processes that may still be running using control-C or:

```
$ killall istioctl
```

- If you are not planning to explore any follow-on tasks, refer to the Bookinfo cleanup instructions to shutdown the application.





# Visualizing Metrics with Grafana

⌚ 4 minute read ✓ page test

---

This task shows you how to setup and use the Istio Dashboard to monitor mesh traffic. As part of this task, you will use the Grafana Istio addon and the web-based interface for viewing service mesh traffic data.

The Bookinfo sample application is used as the example application throughout this task.

# Before you begin

- Install Istio in your cluster.
- Install the Grafana Addon.
- Install the Prometheus Addon.
- Deploy the Bookinfo application.

# Viewing the Istio dashboard

1. Verify that the `prometheus` service is running in your cluster.

In Kubernetes environments, execute the following command:

```
$ kubectl -n istio-system get svc prometheus
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
prometheus    ClusterIP   10.100.250.202  <none>       9090/TCP
              103s
```

2. Verify that the Grafana service is running in your

cluster.

In Kubernetes environments, execute the following command:

```
$ kubectl -n istio-system get svc grafana
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)
)        AGE
grafana   ClusterIP  10.103.244.103  <none>          3000/TCP
CP      2m25s
```

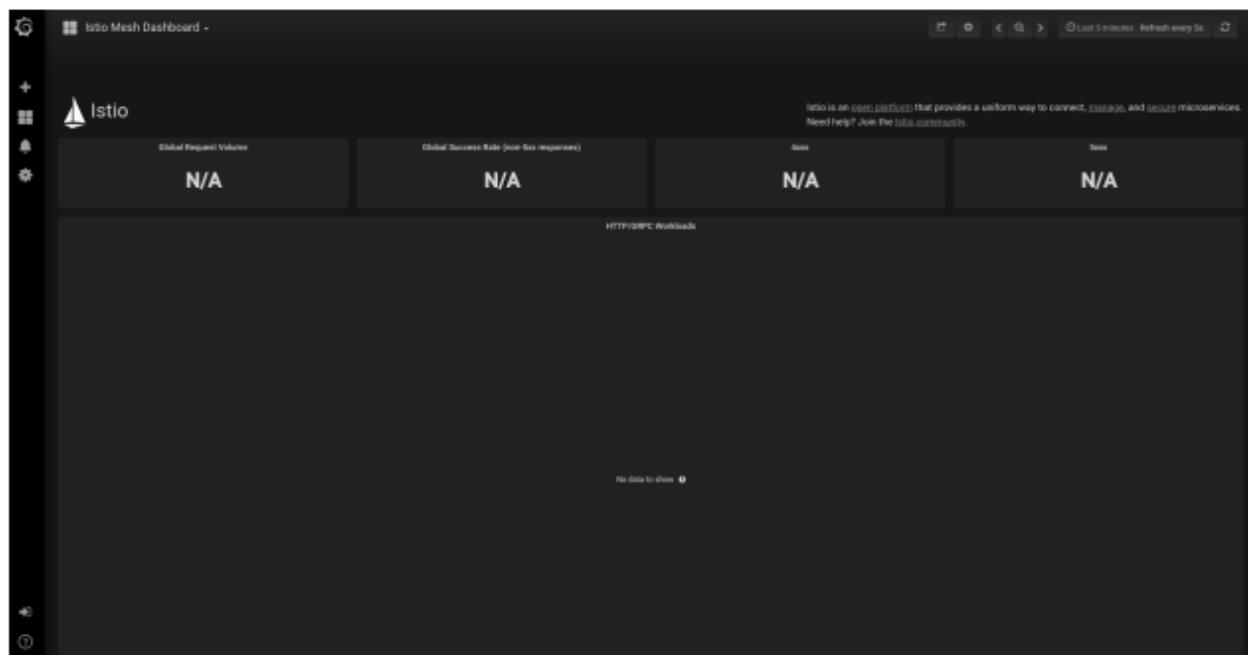
### 3. Open the Istio Dashboard via the Grafana UI.

In Kubernetes environments, execute the following command:

```
$ istioctl dashboard grafana
```

**Visit <http://localhost:3000/dashboard/db/istio-mesh> dashboard in your web browser.**

The Istio Dashboard will look similar to:



## 4. Send traffic to the mesh.

For the Bookinfo sample, visit

`http://$GATEWAY_URL/productpage` in your web browser or issue the following command:

To see trace data, you must send requests to your service. The number of requests depends on Istio's sampling rate. You set this rate when you install Istio. The default sampling rate is 1%. You need to send at least 100 requests before the first trace is visible. To send a 100 requests to the

productpage service, use the following command:

```
$ for i in $(seq 1 100); do curl -s -o /dev/null "http://$GATEWAY_URL/productpage"; done
```



\$GATEWAY\_URL is the value set in the Bookinfo example.

Refresh the page a few times (or send the command a few times) to generate a small amount of traffic.

Look at the Istio Dashboard again. It should

reflect the traffic that was generated. It will look similar to:

Istio Mesh Dashboard

Last 5 minutes Refresh every 10s

# Istio

Global Request Volume: 1.4 ops

Global Success Rate (non-fail responses): 100%

Latency: N/A

Errors: N/A

Istio is an open platform that provides a uniform way to connect, manage, and secure microservices. Need help? Join the [Istio community](#).

HTTP/GRPC Workloads						
Service	Workload	Requests	P90 Latency	P95 Latency	P99 Latency	Success Rate
reviews.defaultsvc.cluster.local	reviews-v3.default	0.02 ops	18 ms	23 ms	28 ms	100.00%
reviews.defaultsvc.cluster.local	reviews-v2.default	0.05 ops	21 ms	46 ms	50 ms	100.00%
reviews.defaultsvc.cluster.local	reviews-v1.default	0.07 ops	8 ms	19 ms	24 ms	100.00%
ratings.defaultsvc.cluster.local	ratings-v1.default	0.07 ops	3 ms	5 ms	5 ms	100.00%
productpage.defaultsvc.cluster.local	productpage-v1.default	0.15 ops	25 ms	60 ms	96 ms	100.00%
istio-instrumentation.istio-system.svc.cluster.local	istio-instrumentation.istio-system	0.05 ops	3 ms	7 ms	10 ms	100.00%
istio-policy.istio-system.svc.cluster.local	istio-policy.istio-system	0.25 ops	3 ms	5 ms	5 ms	100.00%
details.defaultsvc.cluster.local	details-v1.default	0.15 ops	3 ms	5 ms	5 ms	100.00%

## Istio Dashboard With Traffic

This gives the global view of the Mesh along with services and workloads in the mesh. You can get more details about services and workloads by navigating to their specific dashboards as explained below.

### 5. Visualize Service Dashboards.

From the Grafana dashboard's left hand corner navigation menu, you can navigate to Istio Service Dashboard or visit

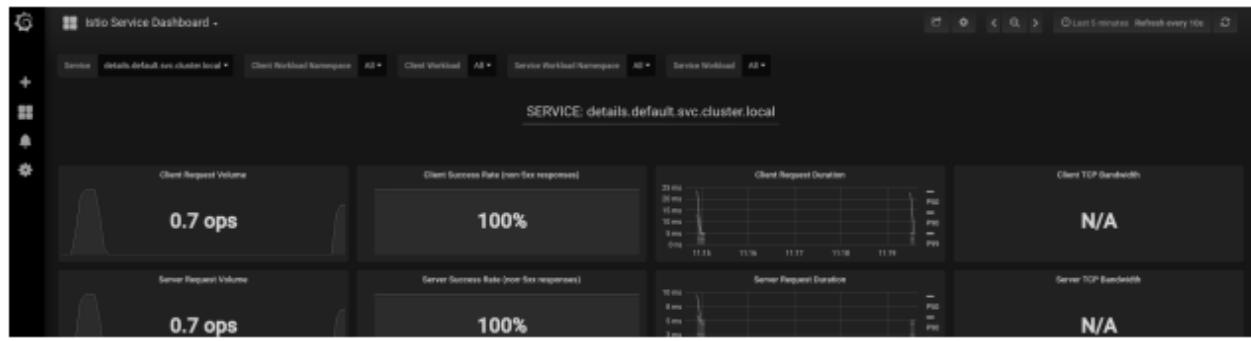
<http://localhost:3000/dashboard/db/istio-service->

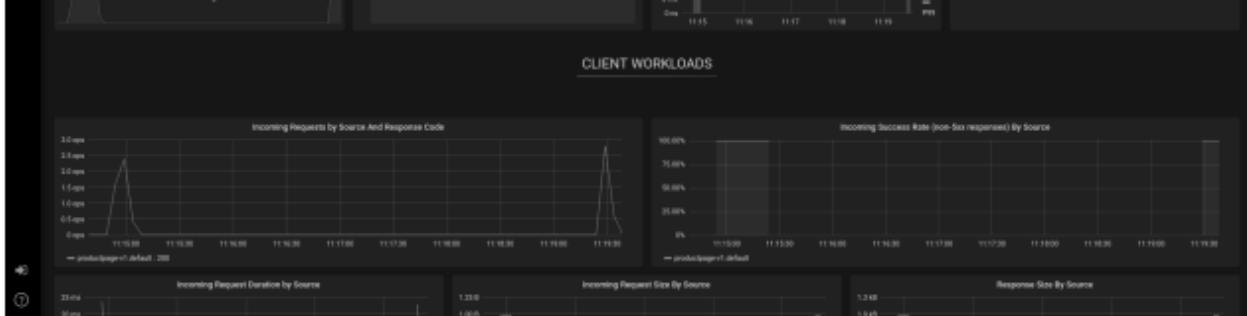
dashboard in your web browser.



You may need to select a service in the Service dropdown.

The Istio Service Dashboard will look similar to:





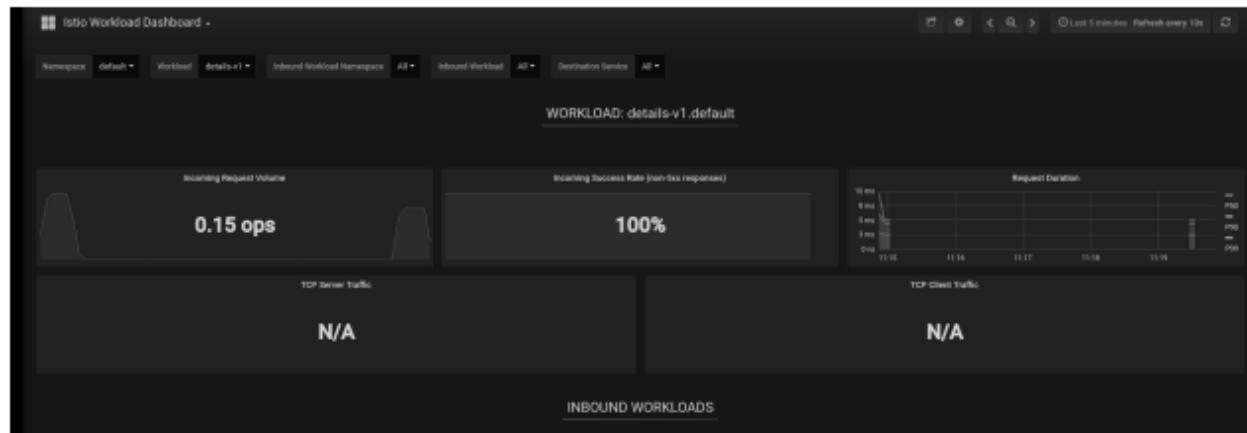
## Istio Service Dashboard

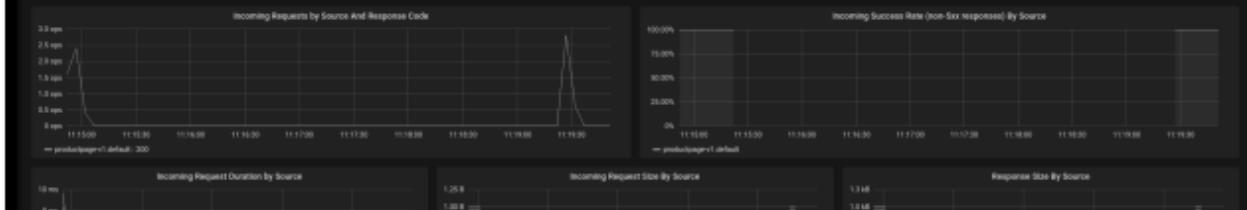
This gives details about metrics for the service and then client workloads (workloads that are calling this service) and service workloads (workloads that are providing this service) for that service.

### 6. Visualize Workload Dashboards.

From the Grafana dashboard's left hand corner navigation menu, you can navigate to Istio Workload Dashboard or visit <http://localhost:3000/dashboard/db/istio-workload-dashboard> in your web browser.

The Istio Workload Dashboard will look similar to:





## Istio Workload Dashboard

This gives details about metrics for each workload and then inbound workloads (workloads that are sending request to this workload) and outbound services (services to which this workload send requests) for that workload.

# About the Grafana dashboards

The Istio Dashboard consists of three main sections:

1. A Mesh Summary View. This section provides Global Summary view of the Mesh and shows HTTP/gRPC and TCP workloads in the Mesh.
2. Individual Services View. This section provides metrics about requests and responses for each individual service within the mesh (HTTP/gRPC and TCP). This also provides metrics about client

and service workloads for this service.

3. Individual Workloads View: This section provides metrics about requests and responses for each individual workload within the mesh (HTTP/gRPC and TCP). This also provides metrics about inbound workloads and outbound services for this workload.

For more on how to create, configure, and edit dashboards, please see the Grafana documentation.

# Cleanup

- Remove any kubectl port-forward processes that may be running:

```
$ killall kubectl
```

- If you are not planning to explore any follow-on tasks, refer to the Bookinfo cleanup instructions to shutdown the application.

# Getting Envoy's Access Logs

⌚ 4 minute read ✓ page test

---

The simplest kind of Istio logging is Envoy's access logging. Envoy proxies print access information to their standard output. The standard output of Envoy's containers can then be printed by the `kubectl logs` command.

# Before you begin

- Setup Istio by following the instructions in the Installation guide.



The egress gateway and access logging will be enabled if you install the demo configuration profile.

- Deploy the sleep sample app to use as a test source for sending requests. If you have automatic

sidecar injection enabled, run the following command to deploy the sample app:

```
$ kubectl apply -f @samples/sleep/sleep.yaml@
```

Otherwise, manually inject the sidecar before deploying the sleep application with the following command:

```
$ kubectl apply -f <(istioctl kube-inject -f @samples/sleep/sleep.yaml@)
```



You can use any pod with curl installed as a test source.

- Set the `SOURCE_POD` environment variable to the name of your source pod:

```
$ export SOURCE_POD=$(kubectl get pod -l app=sleep -o jsonpath={.items..metadata.name})
```

- Start the httpbin sample.

If you have enabled automatic sidecar injection, deploy the httpbin service:

```
$ kubectl apply -f @samples/httpbin/httpbin.yaml@
```

Otherwise, you have to manually inject the

sidecar before deploying the `httpbin` application:

```
$ kubectl apply -f <(istioctl kube-inject -f @samples/httpbin/httpbin.yaml@)
```

## Enable Envoy's access logging

If you used an `IstioOperator` CR to install Istio, add the following field to your configuration:

```
spec:  
  meshConfig:  
    accessLogFile: /dev/stdout
```

Otherwise, add the equivalent setting to your original `istioctl install` command, for example:

```
$ istioctl install <flags-you-used-to-install-Istio> --set meshC  
onfig.accessLogFile=/dev/stdout
```

You can also choose between JSON and text by setting `accessLogEncoding` to `JSON` or `TEXT`.

You may also want to customize the format of the access log by editing `accessLogFormat`.

Refer to global mesh options for more information on all three of these settings:

- `meshConfig.accessLogFile`
- `meshConfig.accessLogEncoding`
- `meshConfig.accessLogFormat`

## **Default access log format**

Istio will use the following default access log format if `accessLogFormat` is not specified:

```
[%START_TIME%] \"%REQ( :METHOD)% %REQ(X-ENVOY-ORIGINAL-PATH?:PATH
)% %PROTOCOL%\" %RESPONSE_CODE% %RESPONSE_FLAGS% %RESPONSE_CODE_
DETAILS% %CONNECTION_TERMINATION_DETAILS%
\"%UPSTREAM_TRANSPORT_FAILURE_REASON%\" %BYTES_RECEIVED% %BYTES_
SENT% %DURATION% %RESP(X-ENVOY-UPSTREAM-SERVICE-TIME)% \"%REQ(X-
FORWARDED-FOR)%\" \"%REQ(USER-AGENT)%\" \"%REQ(X-REQUEST-ID)%\""
 \"%REQ( :AUTHORITY)%\" \"%UPSTREAM_HOST%\" %UPSTREAM_CLUSTER% %UP
STREAM_LOCAL_ADDRESS% %DOWNSTREAM_LOCAL_ADDRESS% %DOWNSTREAM_REM
OTE_ADDRESS% %REQUESTED_SERVER_NAME% %ROUTE_NAME%\n
```

The following table shows an example using the default access log format for a request sent from sleep to httpbin:

Log operator	access log in sleep	access log in httpbin

[%START_TIME%]	[2020-11-25T21:26:18.409Z]	[2020-11-25T21:26:18.409Z]
\">%REQ(:METHOD)% %REQ(X-ENVOY-ORIGINAL-PATH?:PATH)% %PROTOCOL%\\"	"GET /status/418 HTTP/1.1"	"GET /status/418 HTTP/1.1"
%RESPONSE_CODE%	418	418

%RESPONSE_FLA GS%	-	-
%RESPONSE_COD E_DETAILS%	via_upstream	via_upstream
%CONNECTION_T ERMINATION_DE TAILS%	-	-
\%"UPSTREAM_T RANSPORT_FAIL URE_REASON%"	" - "	" - "
%BYTES_RECEIV	0	0

ED%		
%BYTES_SENT%	135	135
%DURATION%	4	3
%RESP(X- ENVOY- UPSTREAM- SERVICE- TIME)%	4	1
\ "%REQ(X- FORWARDED-	" - "	" - "

FOR)%\\"		
\">%REQ(USER-AGENT)%\"	"curl/7.73.0-DEV"	"curl/7.73.0-DEV"
\">%REQ(X-REQUEST-ID)%\"	"84961386-6d84-929d-98bd-c5aee93b5c88"	"84961386-6d84-929d-c5aee93b5c88"
\">%REQ(:AUTHORITY)%\"	"httpbin:8000"	"httpbin:8000"
\">%UPSTREAM_HOST%\"	"10.44.1.27:80"	"127.0.0.1:80"

%UPSTREAM_CLUSTER%	outbound 8000  http bin.foo.svc.cluster .local	inbound 8000
%UPSTREAM_LOCAL_ADDRESS%	10.44.1.23:37652	127.0.0.1:41854
%DOWNSTREAM_LOCAL_ADDRESS%	10.0.45.184:8000	10.44.1.27:80
%DOWNSTREAM_REMOTE_ADDRESS%	10.44.1.23:46520	10.44.1.23:37652
%REQUESTED_SERVICE%	-	outbound_.8000_

RVER_NAME%		pbin.foo.svc.clus
		local
%ROUTE_NAME%	default	default

## Test the access log

1. Send a request from sleep to httpbin:

```
$ kubectl exec "$SOURCE_POD" -c sleep -- curl -sS -v httpbin:8000/status/418
...
< HTTP/1.1 418 Unknown
< server: envoy
...
-[ teapot ]=-
```



## 2. Check sleep's log:

```
$ kubectl logs -l app=sleep -c istio-proxy
[2020-11-25T21:26:18.409Z] "GET /status/418 HTTP/1.1" 418 -
via_upstream - "-" 0 135 4 4 "-" "curl/7.73.0-DEV" "849613
86-6d84-929d-98bd-c5aeee93b5c88" "httpbin:8000" "10.44.1.27:
80" outbound|8000||httpbin.foo.svc.cluster.local 10.44.1.23:
37652 10.0.45.184:8000 10.44.1.23:46520 - default
```

### 3. Check httpbin's log:

```
$ kubectl logs -l app=httpbin -c istio-proxy
[2020-11-25T21:26:18.409Z] "GET /status/418 HTTP/1.1" 418 -
via_upstream - "-" 0 135 3 1 "-" "curl/7.73.0-DEV" "849613
86-6d84-929d-98bd-c5aeee93b5c88" "httpbin:8000" "127.0.0.1:8
0" inbound|8000|| 127.0.0.1:41854 10.44.1.27:80 10.44.1.23:
37652 outbound_.8000_._.httpbin.foo.svc.cluster.local defau
lt
```

Note that the messages corresponding to the request

appear in logs of the Istio proxies of both the source and the destination, sleep and httpbin, respectively. You can see in the log the HTTP verb (GET), the HTTP path (/status/418), the response code (418) and other request-related information.

## Cleanup

Shutdown the sleep and httpbin services:

```
$ kubectl delete -f @samples/sleep/sleep.yaml@  
$ kubectl delete -f @samples/httpbin/httpbin.yaml@
```

# Disable Envoy's access logging

Remove, or set to "", the `meshConfig.accessLogFile` setting in your Istio install configuration.

In the example below, replace `default` with the name of the profile you used when you

## installed Istio.

```
$ istioctl install --set profile=default
✓ Istio core installed
✓ Istiod installed
✓ Ingress gateways installed
✓ Installation complete
```

# Overview

⌚ 2 minute read

---

Distributed tracing enables users to track a request through mesh that is distributed across multiple services. This allows a deeper understanding about request latency, serialization and parallelism via visualization.

Istio leverages Envoy's distributed tracing **feature** to

provide tracing integration out of the box. Specifically, Istio provides options to install various tracing backend and configure proxies to send trace spans to them automatically. See [Zipkin](#), [Jaeger](#) and [Lightstep](#) task docs about how Istio works with those tracing systems.

## Trace context propagation

Although Istio proxies are able to automatically send spans, they need some hints to tie together the entire

trace. Applications need to propagate the appropriate HTTP headers so that when the proxies send span information, the spans can be correlated correctly into a single trace.

To do this, an application needs to collect and propagate the following headers from the incoming request to any outgoing requests:

- x-request-id
- x-b3-traceid
- x-b3-spanid
- x-b3-parentspanid

- x-b3-sampled
- x-b3-flags
- x-ot-span-context

Additionally, tracing integrations based on OpenCensus (e.g. Stackdriver) propagate the following headers:

- x-cloud-trace-context
- traceparent
- grpc-trace-bin

If you look at the sample Python productpage service, for example, you see that the application extracts the

required headers from an HTTP request using  
OpenTracing libraries:

```
def getForwardHeaders(request):
    headers = {}

    # x-b3-*** headers can be populated using the opentracing sp
an
    span = get_current_span()
    carrier = {}
    tracer.inject(
        span_context=span.context,
        format=Format.HTTP_HEADERS,
        carrier=carrier)

    headers.update(carrier)

    # ...
```

```
incoming_headers = ['x-request-id', 'x-datadog-trace-id', 'x-
-datadog-parent-id', 'x-datadog-sampled']

# ...

for ihdr in incoming_headers:
    val = request.headers.get(ihdr)
    if val is not None:
        headers[ihdr] = val

return headers
```

The reviews application (Java) does something similar using `requestHeaders`:

```
@GET  
@Path("/reviews/{productId}")  
public Response bookReviewsById(@PathParam("productId") int productId, @Context HttpHeaders requestHeaders) {  
  
    // ...  
  
    if (ratings_enabled) {  
        JsonObject ratingsResponse = getRatings(Integer.toString(productId), requestHeaders);
```

When you make downstream calls in your applications, make sure to include these headers.

# Configuring tracing using the Telemetry API

⌚ 5 minute read  page test

---

---

Istio provides a Telemetry API that enables flexible configuration of tracing behavior. The Telemetry API offers control over tracing options such as sampling rates and custom tags for individual spans, as well as

backend provider selection.

# Before you begin

1. Ensure that your applications propagate tracing headers.
2. Follow the tracing installation guide located under Integrations to install your preferred tracing provider.

# Telemetry API: Tracing Overview

The Telemetry API offers tracing behavior configuration control over the following at the mesh, namespace, and workload levels:

- **provider selection** - allows selection of backend providers for reporting.
- **sampling percentage** - allows control of the rate of trace sampling applied to received requests *for which no prior sampling decision has been made.*

- **custom tags** - allows control over any custom tags to add to each generated tracing span.
- **tracing participation** - allows opting services out of reporting spans to the selected tracing providers.

## Workload Selection

Individual workloads within a namespace are selected via a selector which allows label-based selection of workloads.

It is not valid to have two different Telemetry resources select the same workload using selector. Likewise, it is not valid to have two distinct Telemetry resources in a namespace with no selector specified.

## Scope, Inheritance, and Overrides

Telemetry API resources inherit configuration from parent resources in the Istio configuration hierarchy:

1. root configuration namespace (example: istio-

- system)
2. local namespace (namespace-scoped resource with **no** workload selector)
  3. workload (namespace-scoped resource with a workload selector)

A Telemetry API resource in the root configuration namespace, typically `istio-system`, provides mesh-wide defaults for behavior. Any workload-specific selector in the root configuration namespace will be ignored/rejected. It is not valid to define multiple mesh-wide Telemetry API resources in the root configuration namespace.

Namespace-specific overrides for the mesh-wide configuration can be achieved by applying a new Telemetry resource in the desired namespace (without a workload selector). Any Tracing fields specified in the namespace configuration will completely override the field from the parent configuration (in the root configuration namespace).

Workload-specific overrides can be achieved by applying a new Telemetry resource in the desired namespace *with a workload selector*. Any Tracing fields specified in the namespace configuration will completely override the field from any parent configuration (root configuration or local namespace).

# Using the Telemetry API for Tracing Configuration

## Configuring tracing providers

Tracing providers are the backend collectors and processors that receive tracing spans and process them for storage and retrieval. Example providers include Zipkin, Jaeger, Lightstep, Datadog, and Apache SkyWalking.

For Istio, tracing providers are configured for use within the mesh via `MeshConfig`. To configure new providers to use in tracing, edit the `MeshConfig` for your mesh via:

```
$ kubectl -n istio-system edit configmap istio
```

The full set of configuration options is described in the reference docs for `MeshConfig`. Typical configuration includes service address and port for the provider, as well as establishing a limit on max tag length supported by the provider.

Each configured provider *must* be uniquely named.

That name will be used to refer to the provider in the Telemetry API.

An example set of provider configuration in `MeshConfig` is:

```
data:  
  mesh: |-  
    extensionProviders: # The following content defines two ex  
ample tracing providers.  
    - name: "localtrace"  
      zipkin:  
        service: "zipkin.istio-system.svc.cluster.local"  
        port: 9411  
        maxTagLength: 56  
    - name: "cloudtrace"  
      stackdriver:  
        maxTagLength: 256  
      defaultProviders: # If a default provider is not specified  
, Telemetry resources must fully-specify a provider  
        tracing: "cloudtrace"
```

# Configuring mesh-wide

# tracing behavior

Telemetry API resources inherit from the root configuration namespace for a mesh, typically `istio-system`. To configure mesh-wide behavior, add a new (or edit the existing) `Telemetry` resource in the root configuration namespace.

Here is an example configuration that uses the provider configuration from the prior section:

```
apiVersion: telemetry.istio.io/v1alpha1
kind: Telemetry
metadata:
  name: mesh-default
  namespace: istio-system
spec:
  tracing:
    - providers: # only a single tracing provider is supported at
this time
      - name: localtrace
  customTags:
    foo:
      literal:
        value: bar
  randomSamplingPercentage: 100
```

This configuration overrides the default provider from MeshConfig, setting the mesh default to be the

"localtrace" provider. It also sets the mesh-wide sampling percentage to be 100, and configures a tag to be added to all trace spans with a name of `foo` and a value of `bar`.

## Configuring namespace-scoped tracing behavior

To tailor the tracing behavior for individual namespaces, add a `Telemetry` resource to the desired namespace. Any tracing fields specified in the namespace resource will completely override the

inherited field configuration from the configuration hierarchy. For example:

```
apiVersion: telemetry.istio.io/v1alpha1
kind: Telemetry
metadata:
  name: namespace-override
  namespace: myapp
spec:
  tracing:
  - customTags:
    userId:
      header:
        name: userId
        defaultValue: unknown
```

When deployed with into a mesh with the prior mesh-

wide example configuration, this will result in tracing behavior in the `myapp` namespace that sends trace spans to the `localtrace` provider and randomly selects requests for tracing at a `100%` rate, but that sets custom tags for each span with a name of `userId` and a value taken from the `userId` request header.

Importantly, the `foo: bar` tag from the parent configuration will not be used in the `myapp` namespace. The custom tags behavior completely overrides the behavior configured in the `mesh-default.istio-system` resource.

①

Any tracing configuration in a Telemetry resource completely overrides configuration of its parent resource in the configuration hierarchy. This includes provider selection.

## Configuring workload-specific tracing behavior

To tailor the tracing behavior for individual

workloads, add a `Telemetry` resource to the desired namespace and use a `selector`. Any tracing fields specified in the workload-specific resource will completely override the inherited field configuration from the configuration hierarchy.

For example:

```
apiVersion: telemetry.istio.io/v1alpha1
kind: Telemetry
metadata:
  name: workload-override
  namespace: myapp
spec:
  selector:
    matchLabels:
      service.istio.io/canonical-name: frontend
  tracing:
    - disableSpanReporting: true
```

In this case, tracing will be disabled for the `frontend` workload in the `myapp` namespace. Istio will still forward the tracing headers, but no spans will be reported to the configured tracing provider.

It is not valid to have two Telemetry resources with workload selectors select the same workload. In those cases, Istio tracing behavior is undefined.

# Configure tracing using MeshConfig and Pod annotations

⌚ 5 minute read  page test

---

---



Users are encouraged to transition to the Telemetry API for tracing configuration.

Istio provides the ability to configure advanced tracing options, such as sampling rate and adding custom tags to reported spans. Sampling is a beta feature, but adding custom tags and tracing tag length are considered in-development for this release.

## **Before you begin**

1. Ensure that your applications propagate tracing headers as described [here](#).

2. Follow the tracing installation guide located under Integrations based on your preferred tracing backend to install the appropriate addon and configure your Istio proxies to send traces to the tracing deployment.

## **Available tracing configurations**

You can configure the following tracing options in Istio:

1. Random sampling rate for percentage of requests that will be selected for trace generation.
2. Maximum length of the request path after which the path will be truncated for reporting. This can be useful in limiting trace data storage specially if you're collecting traces at ingress gateways.
3. Adding custom tags in spans. These tags can be added based on static literal values, environment values or fields from request headers. This can be used to inject additional information in spans specific to your environment.

There are two ways you can configure tracing

options:

1. Globally via `MeshConfig` options.
2. Per-pod annotations for workload specific customization.

 In order for the new tracing configuration to take effect for either of these options you need to restart pods injected with Istio proxies.

Any pod annotations added for tracing configuration override global settings. In order to preserve any global settings you should copy them from global mesh config to pod annotations along with workload specific customization. In particular, make sure that the tracing backend address is always provided in the annotations to ensure that the traces are reported correctly for the workload.

# Installation

Using these features opens new possibilities for managing traces in your environment.

In this example, we will sample all traces and add a tag named `clusterID` using the `ISTIO_META_CLUSTER_ID` environment variable injected into your pod. Only the first 256 characters of the value will be used.

```
$ cat <<EOF > ./tracing.yaml
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  meshConfig:
    enableTracing: true
    defaultConfig:
      tracing:
        sampling: 100.0
        max_path_tag_length: 256
        custom_tags:
          clusterID:
          environment:
            name: ISTIO_META_CLUSTER_ID
EOF
$ istioctl install -f ./tracing.yaml
```

# Using MeshConfig for trace settings

All tracing options can be configured globally via MeshConfig. To simplify configuration, it is recommended to create a single YAML file which you can pass to the `istioctl install -f` command.

```
cat <<'EOF' > tracing.yaml
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  meshConfig:
    enableTracing: true
    defaultConfig:
      tracing:
        sampling: 10
        custom_tags:
          my_tag_header:
            header:
              name: host
EOF
```

# Using proxy.istio.io/config

# annotation for trace settings

You can add the `proxy.istio.io/config` annotation to your Pod metadata specification to override any mesh-wide tracing settings. For instance, to modify the `sleep` deployment shipped with Istio you would add the following to `samples/sleep/sleep.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sleep
spec:
  ...
  template:
    metadata:
      ...
      proxy.istio.io/config: |
        tracing:
          sampling: 10
        custom_tags:
          my_tag_header:
            header:
              name: host
spec:
  ...
```

# Customizing Trace sampling

The sampling rate option can be used to control what percentage of requests get reported to your tracing system. This should be configured depending upon your traffic in the mesh and the amount of tracing data you want to collect. The default rate is 1%.

Previously, the recommended method was to change the `values.pilot.traceSampling` setting

⚠ during the mesh setup or to change the `PILOT_TRACE_SAMPLE` environment variable in the pilot or istiod deployment. While this method to alter sampling continues to work, the following method is strongly recommended instead.

In the event that both are specified, the value specified in the `MeshConfig` will override any other setting.

To modify the default random sampling to 50, add the following option to your `tracing.yaml` file.

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  meshConfig:
    enableTracing: true
    defaultConfig:
      tracing:
        sampling: 50
```

The sampling rate should be in the range of 0.0 to 100.0 with a precision of 0.01. For example, to trace 5 requests out of every 10000, use 0.05 as the value here.

# Customizing tracing tags

Custom tags can be added to spans based on literals, environmental variables and client request headers in order to provide additional information in spans specific to your environment.



There is no limit on the number of custom tags that you can add, but tag names must be unique.

You can customize the tags using any of the three supported options below.

1. Literal represents a static value that gets added to each span.

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  meshConfig:
    enableTracing: true
    defaultConfig:
      tracing:
        custom_tags:
          my_tag_literal:
            literal:
              value: <VALUE>
```

2. Environmental variables can be used where the value of the custom tag is populated from a workload proxy environment variable.

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  meshConfig:
    enableTracing: true
    defaultConfig:
      tracing:
        custom_tags:
          my_tag_env:
            environment:
              name: <ENV_VARIABLE_NAME>
              defaultValue: <VALUE>      # optional
```

In order to add custom tags based on environmental variables, you must modify the `istio-sidecar-injector` ConfigMap in your root Istio system namespace.



3. Client request header option can be used to populate tag value from an incoming client request header.

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  meshConfig:
    enableTracing: true
    defaultConfig:
      tracing:
        custom_tags:
          my_tag_header:
            header:
              name: <CLIENT-HEADER>
              defaultValue: <VALUE>           # optional
```

# Customizing tracing tag length

By default, the maximum length for the request path included as part of the `HttpUrl` span tag is 256. To modify this maximum length, add the following to your `tracing.yaml` file.

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  meshConfig:
    enableTracing: true
    defaultConfig:
      tracing:
        max_path_tag_length: <VALUE>
```

# Jaeger

⌚ 2 minute read ✓ page test

---

After completing this task, you understand how to have your application participate in tracing with Jaeger, regardless of the language, framework, or platform you use to build your application.

This task uses the Bookinfo sample as the example application.

To learn how Istio handles tracing, visit this task's overview.

## Before you begin

1. Follow the Jaeger installation documentation to deploy Jaeger into your cluster.
2. When you enable tracing, you can set the sampling rate that Istio uses for tracing. Use the `meshConfig.defaultConfig.tracing.sampling` option during installation to set the sampling rate. The

default sampling rate is 1%.

3. Deploy the Bookinfo sample application.

## Accessing the dashboard

Remotely Accessing Telemetry Addons [details](#) how to configure access to the Istio addons through a gateway.

For testing (and temporary access), you may also use port-forwarding. Use the following, assuming you've

deployed Jaeger to the `istio-system` namespace:

```
$ istioctl dashboard jaeger
```

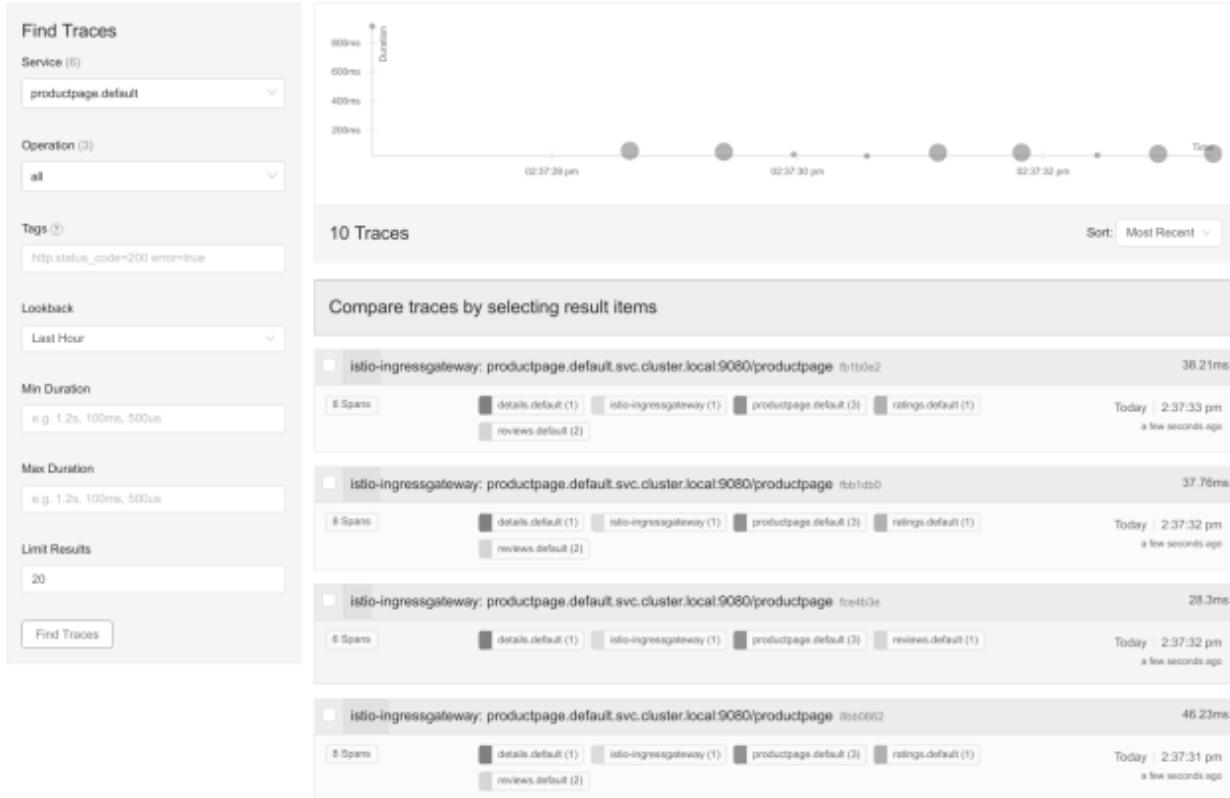
## Generating traces using the Bookinfo sample

1. When the Bookinfo application is up and running, access `http://$GATEWAY_URL/productpage` one or more times to generate trace information.

To see trace data, you must send requests to your service. The number of requests depends on Istio's sampling rate. You set this rate when you install Istio. The default sampling rate is 1%. You need to send at least 100 requests before the first trace is visible. To send 100 requests to the productpage service, use the following command:

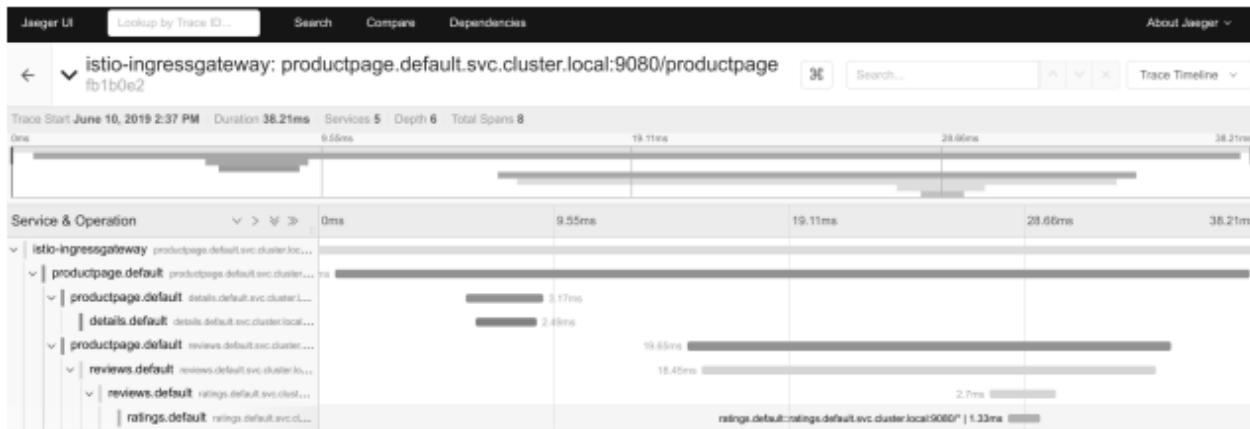
```
$ for i in $(seq 1 100); do curl -s -o /dev/null "http://$G  
ATEWAY_URL/productpage"; done
```

2. From the left-hand pane of the dashboard, select productpage.default from the **Service** drop-down list and click **Find Traces**:



# Tracing Dashboard

3. Click on the most recent trace at the top to see the details corresponding to the latest request to the /productpage:



Detailed Trace View

4. The trace is comprised of a set of spans, where each span corresponds to a Bookinfo service, invoked during the execution of a /productpage request, or internal Istio component, for example: istio-ingressgateway.

## Cleanup

1. Remove any `istioctl` processes that may still be running using control-C or:

```
$ killall istioctl
```

2. If you are not planning to explore any follow-on tasks, refer to the Bookinfo cleanup instructions to shutdown the application.

# Zipkin

⌚ 2 minute read ✓ page test

---

After completing this task, you understand how to have your application participate in tracing with Zipkin, regardless of the language, framework, or platform you use to build your application.

This task uses the Bookinfo sample as the example application.

To learn how Istio handles tracing, visit this task's overview.

## Before you begin

1. Follow the Zipkin installation documentation to deploy Zipkin into your cluster.
2. When you enable tracing, you can set the sampling rate that Istio uses for tracing. Use the `meshConfig.defaultConfig.tracing.sampling` option during installation to set the sampling rate. The

default sampling rate is 1%.

3. Deploy the Bookinfo sample application.

## Accessing the dashboard

Remotely Accessing Telemetry Addons [details](#) how to configure access to the Istio addons through a gateway.

For testing (and temporary access), you may also use port-forwarding. Use the following, assuming you've

deployed Zipkin to the `istio-system` namespace:

```
$ istioctl dashboard zipkin
```

## Generating traces using the Bookinfo sample

1. When the Bookinfo application is up and running, access `http://$GATEWAY_URL/productpage` one or more times to generate trace information.

To see trace data, you must send requests to your service. The number of requests depends on Istio's sampling rate. You set this rate when you install Istio. The default sampling rate is 1%. You need to send at least 100 requests before the first trace is visible. To send 100 requests to the productpage service, use the following command:

```
$ for i in $(seq 1 100); do curl -s -o /dev/null "http://$G  
ATEWAY_URL/productpage"; done
```

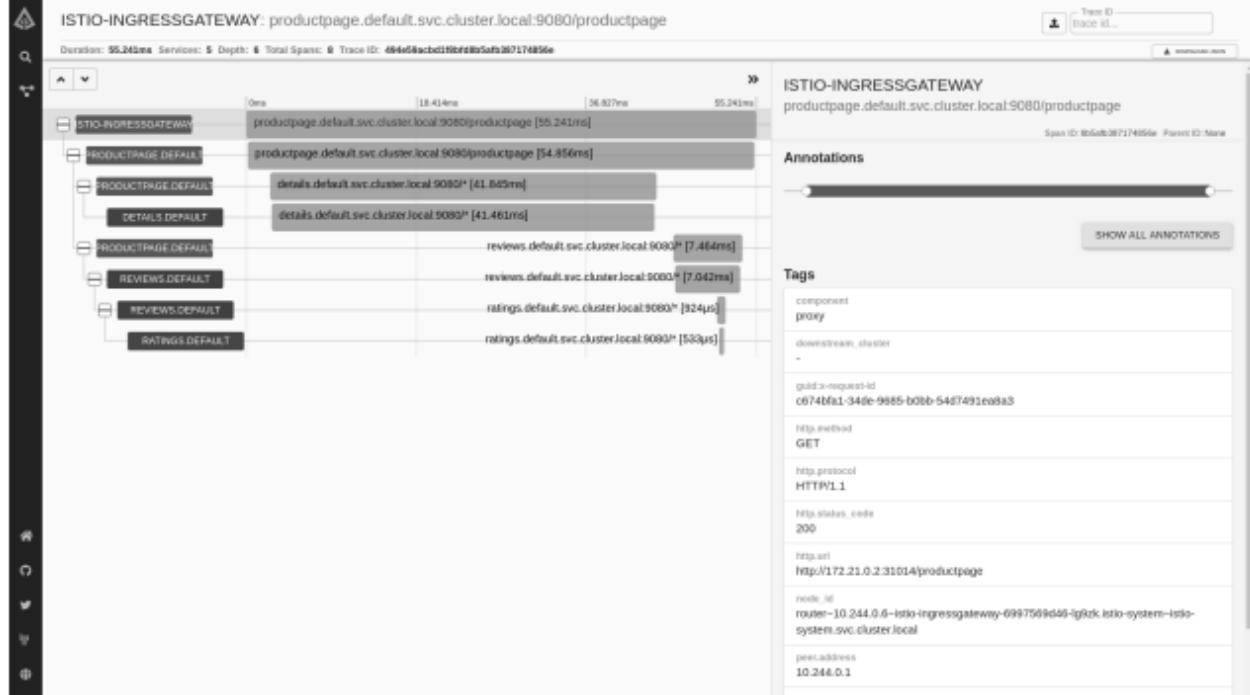
2. From the search panel, click on the plus sign. Select `serviceName` from the first drop-down list, `productpage.default` from second drop-down, and

then click the search icon:

The screenshot shows the Jaeger Tracing Dashboard's 'Discover' interface. At the top, there is a search bar with the placeholder 'There is trace id...'. Below it, a search input field contains the text 'serviceName:productpage default'. To the right of the search input are buttons for '10' and '15 MINUTES', and a magnifying glass icon. A dropdown menu is open next to the search input, showing '1 Result'. The main table has columns: ROOT, TRACE ID, START TIME, and DURATION. One row is shown, corresponding to the search term. The trace ID is listed as 'ISTIO-INGRESSGATEWAY [productpage.default.svc.cluster.local:3080/prod94e650geb1fbfb1b5aef38174855e]'. The start time is '08/11 12:14:29.281 (5 minutes ago)'. The duration is '55.242ms'. Below the table, there is a horizontal navigation bar with five items: 'ISTIO-INGRESSGATEWAY [ ]', 'PRODUCTPAGE DEFAULT [ ]', 'DETAILS DEFAULT [ ]', 'REVIEWS DEFAULT [ ]', and 'RATINGS DEFAULT [ ]'. The first item is highlighted.

## Tracing Dashboard

3. Click on the ISTIO-INGRESSGATEWAY search result to see the details corresponding to the latest request to /productpage:



## Detailed Trace View

4. The trace is comprised of a set of spans, where

each span corresponds to a Bookinfo service, invoked during the execution of a /productpage request, or internal Istio component, for example: `istio-ingressgateway`.

## Cleanup

1. Remove any `istioctl` processes that may still be running using control-C or:

```
$ killall istioctl
```

2. If you are not planning to explore any follow-on tasks, refer to the Bookinfo cleanup instructions to shutdown the application.

# Lightstep

⌚ 5 minute read  page test

---

This task shows you how to configure Istio to collect trace spans and send them to Lightstep. Lightstep lets you analyze 100% of unsampled transaction data from large scale production software to produce meaningful distributed traces and metrics that help explain performance behaviors and accelerate root cause analysis. At the end of this task, Istio sends

trace spans from the proxies to a Lightstep Satellite pool making them available to the web UI. By default, all HTTP requests are captured (to see end-to-end traces, your code needs to forward OT headers even if it does not join the traces).

If you only want to collect tracing spans directly from Istio (and not add specific instrumentation directly to your code), then you don't need to configure any tracers, as long as your services forward the HTTP headers generated by traces.

This task uses the Bookinfo sample application as an example.

# Before you begin

1. Ensure you have a Lightstep account. Sign up for a free trial of Lightstep.
2. If you're using on-premise Satellites, ensure you have a satellite pool configured with TLS certs and a secure GRPC port exposed. See [Install and Configure Satellites](#) for details about setting up satellites.

For Lightstep Public Satellites or Developer Satellites, your satellites are already configured. However you need to download this certificate to a local

directory.

3. Ensure sure you have a Lightstep access token.  
Access tokens allow your app to communicate with your Lightstep project.

## Deploy Istio

How you deploy Istio depends on which type of Satellite you use.

# Deploy Istio with On-Premise Satellites

These instructions do not assume TLS. If you are using TLS for your Satellite pool, follow the config for the Public Satellite pool, but use your own cert and your own pool's endpoint (`host:port`).

1. You need to deploy Istio with your Satellite address at an address in the format `<Host>:<Port>`, for example `lightstep-satellite.lightstep:9292`. You find this in your configuration file.

2. Deploy Istio with the following configuration parameters specified:

- pilot.traceSampling=100
- global.proxy.tracer="lightstep"
- global.tracer.lightstep.address=""
- global.tracer.lightstep.accessToken=""

You can set these parameters using the `--set key=value` syntax when you run the install command. For example:

```
$ istioctl install \
    --set values.pilot.traceSampling=100 \
    --set values.global.proxy.tracer="lightstep" \
    --set values.global.tracer.lightstep.address("<satellite-address>") \
    --set values.global.tracer.lightstep.accessToken("<access-token>") \
```

# Deploy Istio with Public or Developer Mode Satellites

Follow these steps if you're using the Public or Developer Mode Satellites, or if you're using on-

premise Satellites with a TLS certificate.

1. Store your satellite pool's certificate authority certificate as a secret in the default and `istio-system` namespace, the latter for use by the Istio gateways. Download and use this certificate. If you deploy the Bookinfo application in a different namespace, create the secret in that namespace instead.

```
$ CACERT=$(cat Cert_Auth.crt | base64) # Cert_Auth.crt contains the necessary CACert  
$ NAMESPACE=default
```

```
$ cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Secret
metadata:
  name: lightstep.cacert
  namespace: $NAMESPACE
  labels:
    app: lightstep
type: Opaque
data:
  cacert.pem: $CACERT
EOF
```

2. Deploy Istio with the following configuration parameters specified:

```
global:
  proxy:
    tracer: "lightstep"
```

```
tracer:
  lightstep:
    address: "ingest.lightstep.com:443"
    accessToken: "<access-token>"
meshConfig:
  defaultConfig:
    tracing:
      sampling: 100
      tlsSettings
        mode: "SIMPLE"
        # Specifying ca certificate here will moute `lightstep.cacert` secret volume
        # at all sidecars by default.
        caCertificates="/etc/lightstep/cacert.pem"
components:
  ingressGateways:
    # `lightstep.cacert` secret volume needs to be mount at gateways via k8s overlay.
    - name: istio-ingressgateway
      enabled: true
```

```
k8s:  
  overlays:  
    - kind: Deployment  
      name: istio-ingressgateway  
      patches:  
        - path: spec.template.spec.containers[0].volumeMoun  
ts[-1]  
          value: |  
            name: lightstep-certs  
            mountPath: /etc/lightstep  
            readOnly: true  
        - path: spec.template.spec.volumes[-1]  
          value: |  
            name: lightstep-certs  
            secret:  
              secretName: lightstep.cacert  
              optional: true
```

# Install and run the Bookinfo app

1. Follow the instructions to deploy the Bookinfo sample application.
2. Follow the instructions to create an ingress gateway for the Bookinfo application.
3. To verify the previous step's success, confirm that you set `GATEWAY_URL` environment variable in your shell.
4. Send traffic to the sample application.

```
$ curl http://$GATEWAY_URL/productpage
```

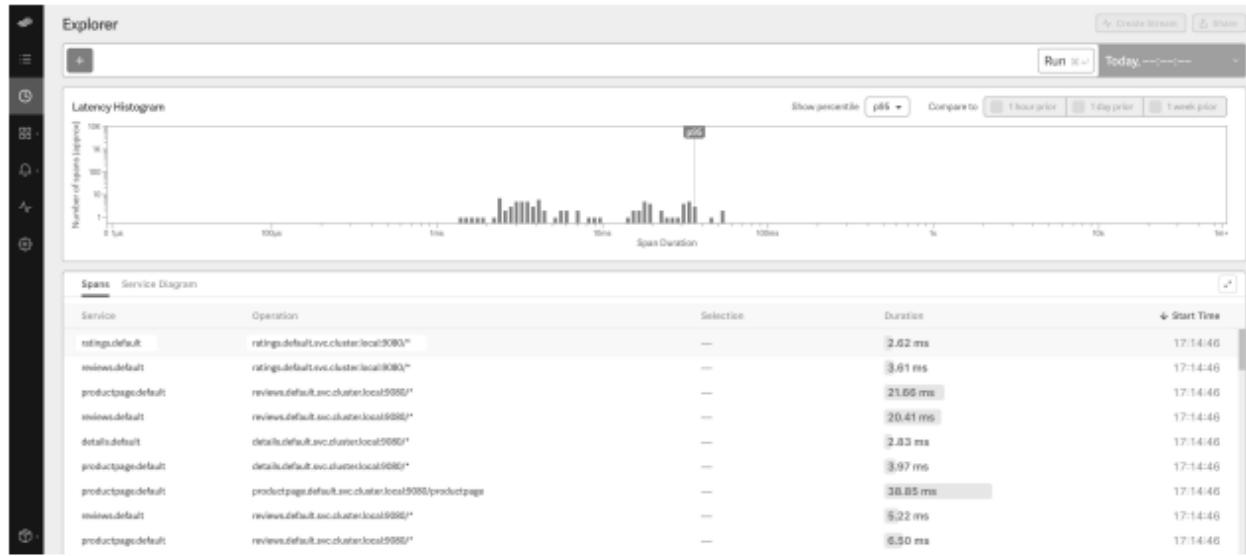
# Visualize trace data

1. Load the Lightstep web UI. You'll see the three Bookinfo services listed in the Service Directory.



# Bookfinder services in the Service Directory

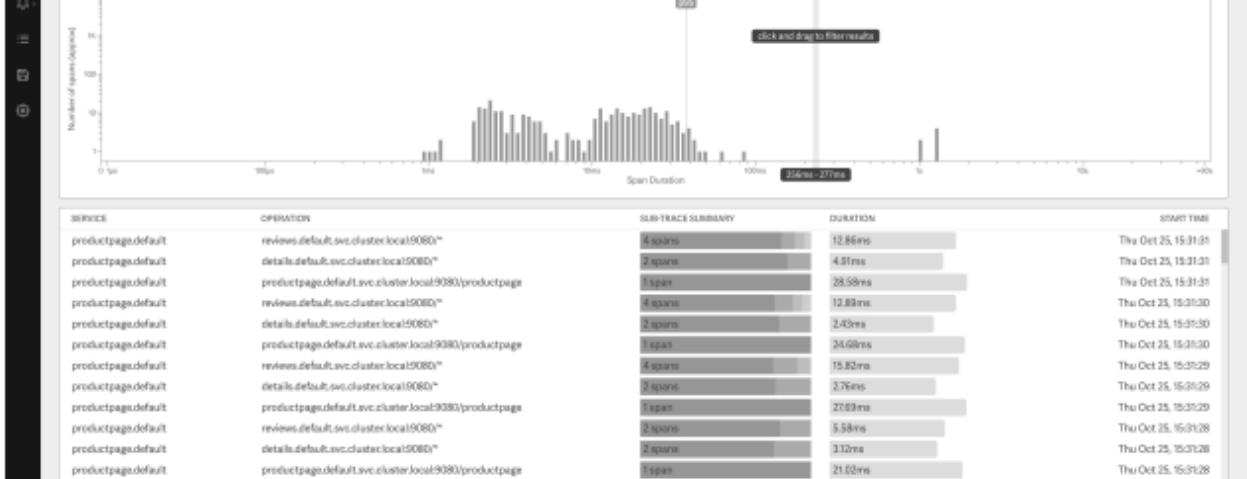
## 2. Navigate to the Explorer view.



## Explorer view

3. Find the query bar at the top. The query bar allows you to interactively filter results by a **Service**, **Operation**, and **Tag** values.
4. Select productpage.default from the **Service** drop-down list.
5. Click **Run**. You see something similar to the following:





## Explorer

- Click on the first row in the table of example traces below the latency histogram to see the details corresponding to your refresh of the /productpage. The page then looks similar to:



## Detailed Trace View

The screenshot shows that the trace is comprised of a set of spans. Each span corresponds to a Bookinfo

service invoked during the execution of a /productpage request.

Two spans in the trace represent every RPC. For example, the call from productpage to reviews starts with the span labeled with the

reviews.default.svc.cluster.local:9080/\* operation and the productpage.default: proxy client service. This service represents the client-side span of the call. The screenshot shows that the call took 15.30 ms. The second span is labeled with the

reviews.default.svc.cluster.local:9080/\* operation and the reviews.default: proxy server service. The second span is a child of the first span and represents the

server-side span of the call. The screenshot shows that the call took 14.60 ms.

## Trace sampling

Istio captures traces at a configurable trace sampling percentage. To learn how to modify the trace sampling percentage, visit the [Distributed Tracing trace sampling section](#).

When using Lightstep, we do not recommend

reducing the trace sampling percentage below 100%. To handle a high traffic mesh, consider scaling up the size of your satellite pool.

## Cleanup

If you are not planning any follow-up tasks, remove the Bookinfo sample application and any Lightstep secrets from your cluster.

1. To remove the Bookinfo application, refer to the

## Bookinfo cleanup instructions.

2. Remove the secret generated for Lightstep:

```
$ kubectl delete secret lightstep.cacert
```

# Visualizing Your Mesh

⌚ 7 minute read  page test

---

This task shows you how to visualize different aspects of your Istio mesh.

As part of this task, you install the Kiali addon and use the web-based graphical user interface to view service graphs of the mesh and your Istio

configuration objects.

This task does not cover all of the features provided by Kiali. To learn about the full set of features it supports, see the Kiali website.

This task uses the Bookinfo sample application as the example throughout. This task assumes the Bookinfo application is installed in the `bookinfo` namespace.

# Before you begin

Follow the Kiali installation documentation to deploy Kiali into your cluster.

## Generating a graph

1. To verify the service is running in your cluster, run the following command:

```
$ kubectl -n istio-system get svc kiali
```

2. To determine the Bookinfo URL, follow the instructions to determine the Bookinfo ingress GATEWAY\_URL.
3. To send traffic to the mesh, you have three options
  - Visit `http://$GATEWAY_URL/productpage` in your web browser
  - Use the following command multiple times:

```
$ curl http://$GATEWAY_URL/productpage
```

- If you installed the `watch` command in your system, send requests continually with:

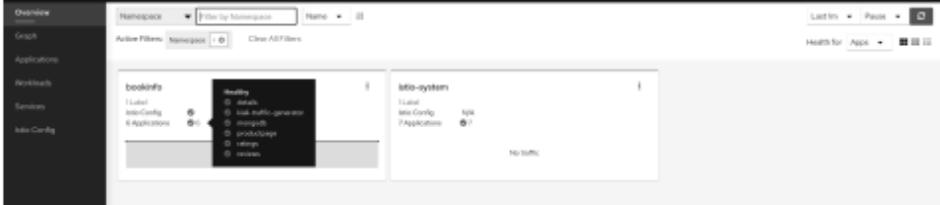
```
$ watch -n 1 curl -o /dev/null -s -w %{http_code} $GATEWAY_URL/productpage
```

- To open the Kiali UI, execute the following command in your Kubernetes environment:

```
$ istioctl dashboard kiali
```

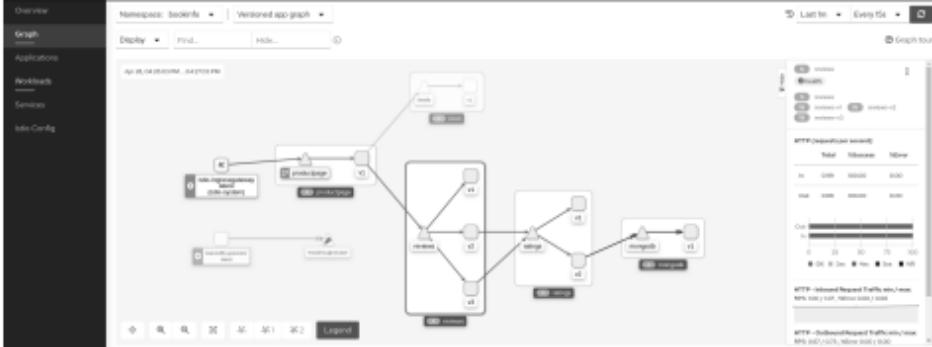
- View the overview of your mesh in the **Overview** page that appears immediately after you log in. The **Overview** page displays all the namespaces that have services in your mesh. The following screenshot shows a similar page:





## Example Overview

6. To view a namespace graph, Select the Graph option in the kebab menu of the Bookinfo overview card. The kebab menu is at the top right of card and looks like 3 vertical dots. Click it to see the available options. The page looks similar to:

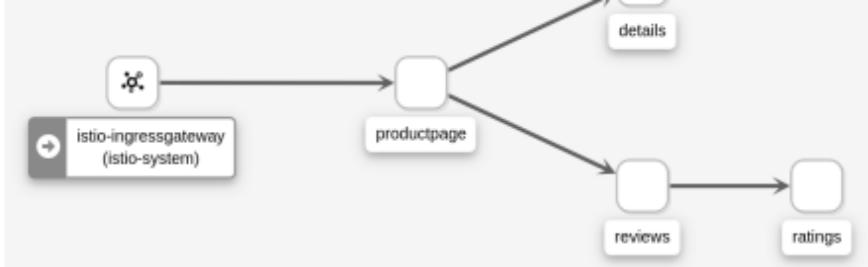


## Example Graph

7. The graph represents traffic flowing through the service mesh for a period of time. It is generated using Istio telemetry.
8. To view a summary of metrics, select any node or edge in the graph to display its metric details in the summary details panel on the right.

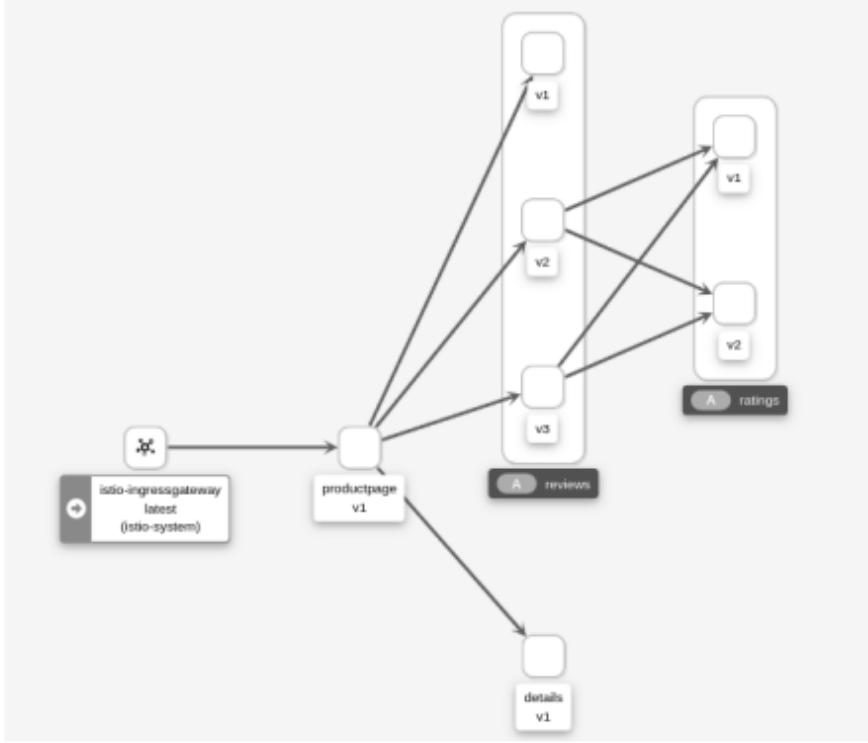
9. To view your service mesh using different graph types, select a graph type from the **Graph Type** drop down menu. There are several graph types to choose from: **App**, **Versioned App**, **Workload**, **Service**.

- The **App** graph type aggregates all versions of an app into a single graph node. The following example shows a single **reviews** node representing the three versions of the reviews app. Note that the **Show Service Nodes** **Display** option has been disabled.



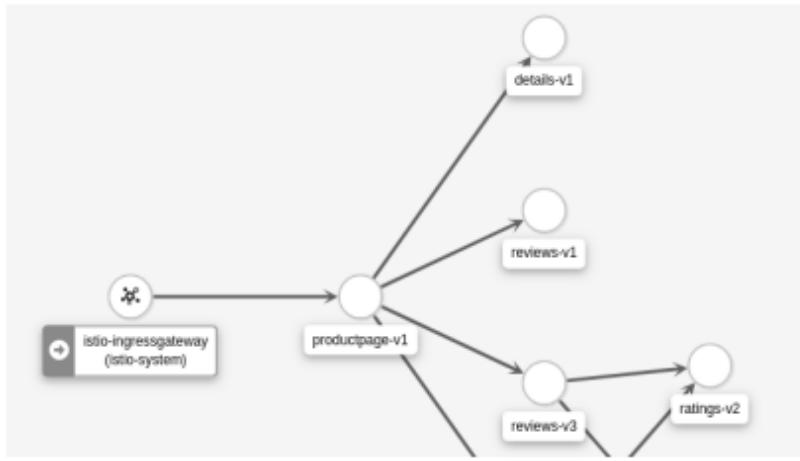
## Example App Graph

- The **Versioned App** graph type shows a node for each version of an app, but all versions of a particular app are grouped together. The following example shows the **reviews** group box that contains the three nodes that represents the three versions of the reviews app.



Example Versioned App Graph

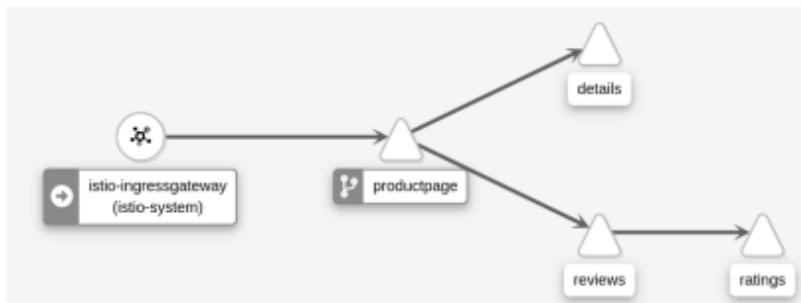
- The **Workload** graph type shows a node for each workload in your service mesh. This graph type does not require you to use the app and version labels so if you opt to not use those labels on your components, this may be your graph type of choice.





## Example Workload Graph

- The **Service** graph type shows a high-level aggregation of service traffic in your mesh.



## Examining Istio configuration

1. The left menu options lead to list views for **Applications**, **Workloads**, **Services** and **Istio Config**. The following screenshot shows **Services** information for the Bookinfo namespace:



The screenshot shows the Kiali UI interface. On the left, there's a sidebar with navigation links: Graph, Applications, View Metrics, Services (which is the selected tab), and Intergit. The main area is titled "Services". At the top, there are filters: "Service Name" and "Filter by Service Name". Below is a table with the following data:

Name	Namespace	Labels	Health	Configuration	Details
details	bookinfo	app:details, service:details	<span>OK</span>	<span>OK</span>	
mongodb	bookinfo	app:mongodb, service:mongodb	<span>OK</span>	<span>OK</span>	
productpage	bookinfo	app:productpage, service:productpage	<span>Healthy</span>	<span>OK</span>	
ratings	bookinfo	app:ratings, service:ratings	<span>OK</span>	<span>OK</span>	
reviews	bookinfo	app:reviews, service:reviews	<span>OK</span>	<span>OK</span>	

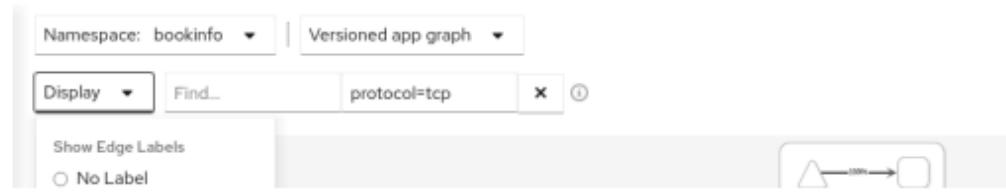
Example Details

# Traffic Shifting

You can use the Kiali traffic shifting wizard to define the specific percentage of request traffic to route to two or more workloads.

1. View the **Versioned app graph** of the bookinfo graph.

- Make sure you have enabled the **Request Distribution Edge Label Display** option to see the percentage of traffic routed to each workload.
- Make sure you have enabled the **Show Service Nodes Display** option to view the service nodes in the graph.



- Request Rate
  - Request Distribution

Show

- Cluster Boxes ⓘ
  - Namespace Boxes ⓘ
  - Compressed Hide ⓘ

Idle Edges ⓘ

Idle Nodes ⓘ

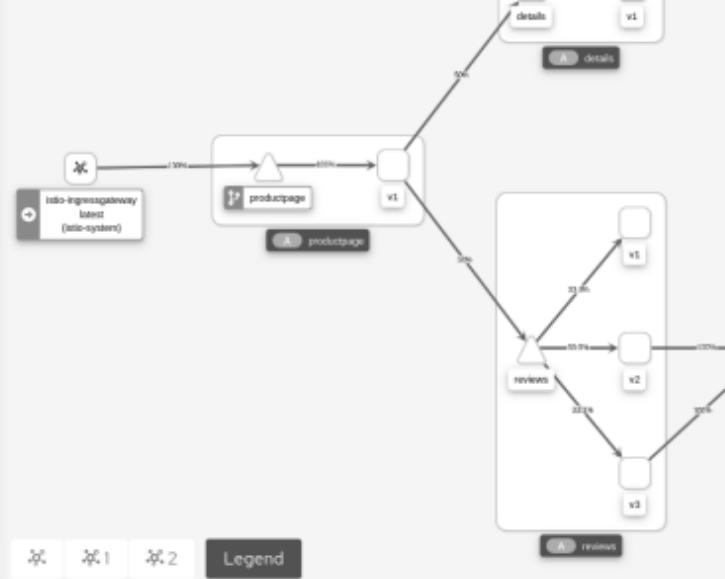
Operation No.

Service Nodes (1)

Traffic Animation ①

## Show Badges

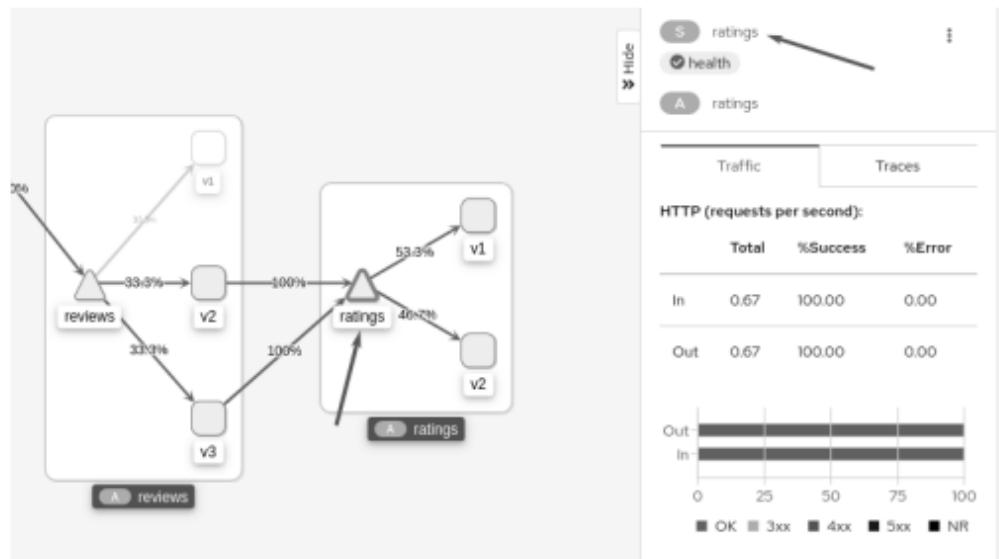
- Circuit Breakers
  - Missing Sidecars
  - Virtual Services



# Bookinfo Graph Options

2. Focus on the ratings service within the bookinfo graph by clicking on the ratings service (triangle) node. Notice the ratings service traffic is evenly

distributed to the two ratings workloads v1 and v2 (50% of requests are routed to each workload).



Graph Showing Percentage of Traffic

3. Click the **ratings** link found in the side panel to go to the detail view for the ratings service. This could also be done by secondary-click on the ratings service node, and selecting Details from the context menu.
4. From the **Actions** drop down menu, select **Traffic Shifting** to access the traffic shifting wizard.





## Service Actions Menu

5. Drag the sliders to specify the percentage of traffic to route to each workload. For `ratings-v1`, set it to 10%; for `ratings-v2` set it to 90%.



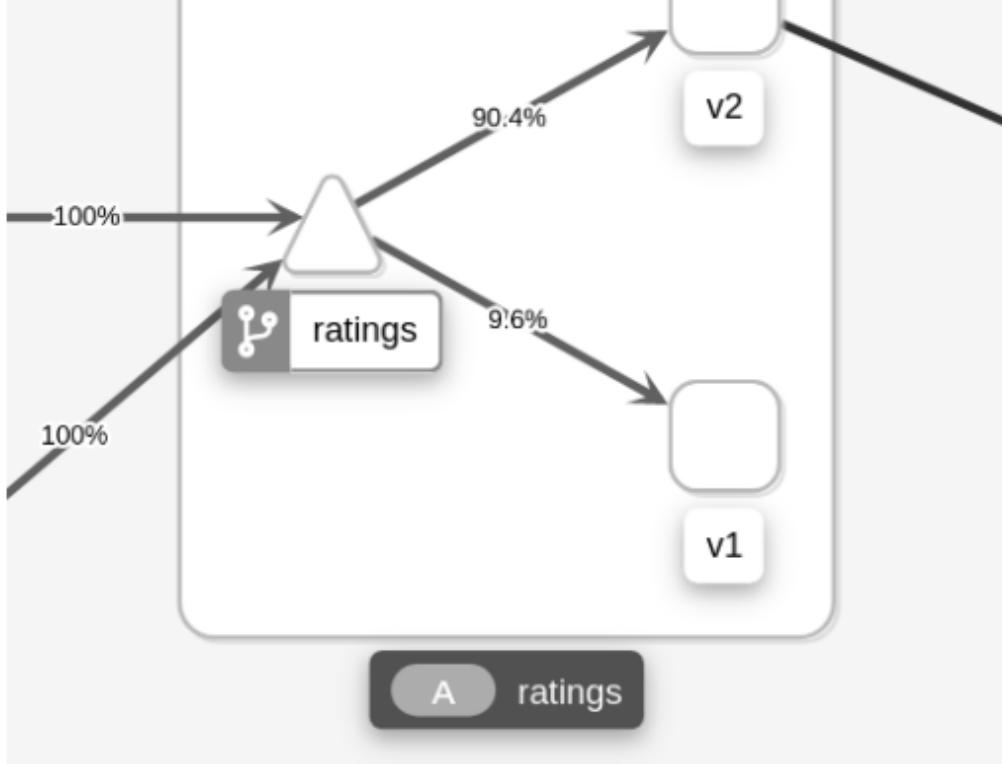
## Weighted Routing Wizard

6. Click the **Create** button to apply the new traffic settings.
7. Click **Graph** in the left hand navigation bar to return to the `bookinfo` graph. Notice that the ratings service node is now badged with the virtual service icon.
8. Send requests to the `bookinfo` application. For example, to send one request per second, you can

execute this command if you have `watch` installed on your system:

```
$ watch -n 1 curl -o /dev/null -s -w %{http_code} $GATEWAY_URL/productpage
```

9. After a few minutes you will notice that the traffic percentage will reflect the new traffic route, thus confirming the fact that your new traffic route is successfully routing 90% of all traffic requests to ratings-v2.



90% Ratings Traffic Routed to  
ratings-v2

# Validating Istio configuration

Kiali can validate your Istio resources to ensure they follow proper conventions and semantics. Any problems detected in the configuration of your Istio resources can be flagged as errors or warnings depending on the severity of the incorrect configuration. See the Kiali validations page for the list of all validation checks Kiali performs.

Istio provides `istioctl analyze` which provides analysis in a way that can be used in a CI pipeline. The two approaches can be complementary.

Force an invalid configuration of a service port name to see how Kiali reports a validation error.

1. Change the port name of the `details` service from `http` to `foo`:

```
$ kubectl patch service details -n bookinfo --type json -p  
'[{"op":"replace","path":"/spec/ports/0/name", "value":"foo"}]'
```

2. Navigate to the **Services** list by clicking **Services** on the left hand navigation bar.
3. Select `bookinfo` from the **Namespace** drop down menu if it is not already selected.
4. Notice the error icon displayed in the **Configuration** column of the details row.

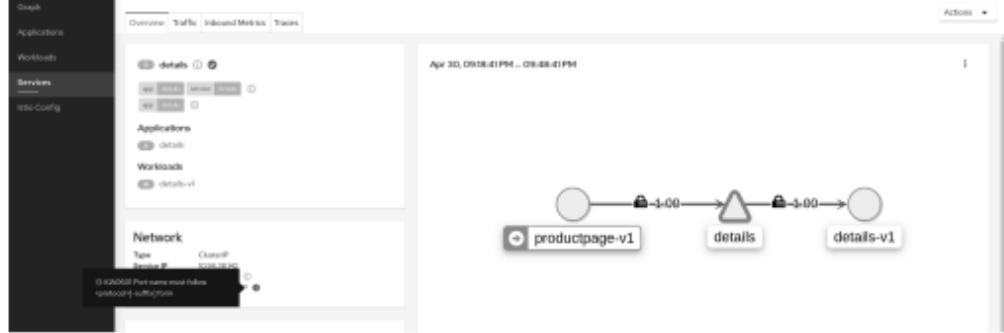


A screenshot of a web application interface. At the top, there's a navigation bar with links for 'Services', 'MongoDB', 'Redis', 'MongoDB + Redis', and 'MongoDB - Redis'. Below the navigation is a search bar with placeholder text 'Search Services' and a magnifying glass icon. The main content area is a table titled 'Services List' with the following data:

Name	Type	Description	Status	Action
mongodb	bookinfo	replica-set - service mongodb		
productpage	bookinfo	replica-set - service productpage		
settings	bookinfo	replica-set - service settings		
reviews	bookinfo	replica-set - service reviews		

## Services List Showing Invalid Configuration

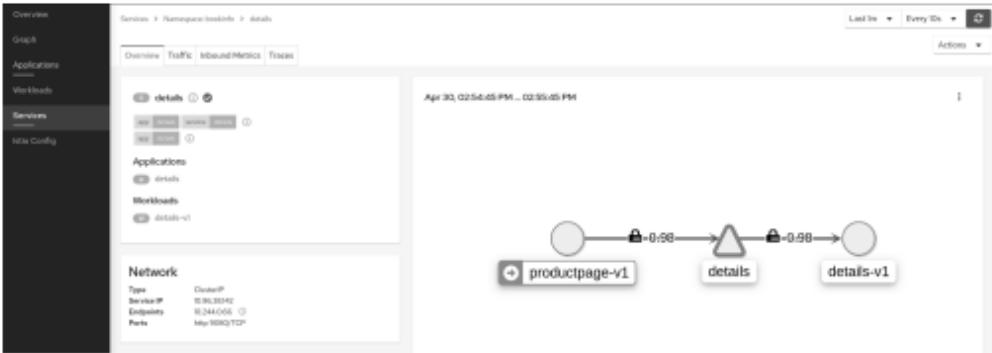
5. Click the **details** link in the **Name** column to navigate to the service details view.
6. Hover over the error icon to display a tool tip describing the error.



## Service Details Describing the Invalid Configuration

7. Change the port name back to `http` to correct the configuration and return `bookinfo` back to its normal state.

```
$ kubectl patch service details -n bookinfo --type json -p  
'[{"op":"replace","path":"/spec/ports/0/name", "value":"http"}]'
```



## Service Details Showing Valid Configuration

# Viewing and editing Istio configuration YAML

Kiali provides a YAML editor for viewing and editing Istio configuration resources. The YAML editor will also provide validation messages when it detects incorrect configurations.

1. Create Bookinfo destination rules:

```
$ kubectl -n bookinfo apply -f @samples/bookinfo/networking  
/destination-rule-all.yaml@
```

2. Click **Istio Config** on the left hand navigation bar

to navigate to the Istio configuration list.

3. Select `bookinfo` from the **Namespace** drop down menu if it is not already selected.
4. Notice the error message and the error icons that alert you to several configuration problems.

Name	Namespace	Type	Configuration
bookinfo	bookinfo	VirtualService	⚠️
bookinfo-gateway	bookinfo	Gateway	⚠️
details	bookinfo	DestinationRule	⚠️
productpage	bookinfo	DestinationRule	⚠️
ratings	bookinfo	VirtualService	⚠️
reviews	bookinfo	DestinationRule	⚠️
service	bookinfo	DestinationRule	⚠️

# Istio Config List Incorrect Configuration Messages

5. Click the error icon in the **Configuration** column of the `details` row to navigate to the `details` destination rule view.
6. The **YAML** tab is preselected. Notice the color highlights and icons on the rows that have failed validation checks.

The screenshot shows the 'Destination Rule Overview' section of the Istio Config interface. It displays a table with one row for a destination rule named 'details'. The table includes columns for 'Name', 'Namespace', 'Labels', 'Annotations', and 'Configuration'. The 'Configuration' column contains an error icon (a red circle with a white exclamation mark) next to the word 'details'. The 'Actions' button is located at the top right of the table. Below the table, there is a 'Destination Rule Overview' section with tabs for 'List' and 'Details'.

Name	Namespace	Labels	Annotations	Configuration
details	bookinfo			details

Destination Rule Overview

List Details



## YAML Editor Showing Validation Errors and Warnings

7. Hover over the red icon to view the tool tip message that informs you of the validation check that triggered the error. For more details on the cause of the error and how to resolve it, look up the validation error message on the Kiali Validations page.

The screenshot shows a YAML editor interface with a code editor window and a toolbar below it.

**Code Editor Content:**

```
1 kind: DestinationRule
2 apiVersion: networking.istio.io/v1alpha3
3 metadata:
4   name: details
5   namespace: bookinfo
6   uid: f0f3031af-a859-4270-b488-902b6a312490
7   resourceVersion: '119860'
8   generation: 1
9   creationTimestamp: '2021-04-30T18:57:55Z'
10 annotations:
11   kubectl.kubernetes.io/last-applied-configuration: >
12     {"apiVersion": "networking.istio.io/v1alpha3", "kind": "DestinationRule", "metadata": {"annotations": {}, "name": "details", "namespace": "bookinfo"}, "spec": {"host": "*.*.bookinfo.svc.cluster.local", "subsets": [{"labels": {"version": "v1"}, "name": "v1"}, {"labels": {"version": "v2"}, "name": "v2"}]}}
13   managedFields: []
14 spec:
15   host: details
16   subsets:
17     - labels:
18       - label:
19         name: v1
20         version: v1
21       - label:
22         name: v2
23         version: v2
24 KLM203 This subset's labels are not found in any matching host
25
26
27
28
29
30
31
32
33
34
35
36
37
```

**Toolbar:**

Save | Reload | Cancel

A red box highlights the error message "KLM203 This subset's labels are not found in any matching host" in the code editor.

## YAML Editor Showing Error Tool Tip

8. Delete the destination rules to return bookinfo back to its original state.

```
$ kubectl -n bookinfo delete -f samples/bookinfo/networking  
/destination-rule-all.yaml
```

# Additional Features

Kiali has many more features than reviewed in this task, such as an integration with Jaeger tracing.

For more details on these additional features, see the Kiali documentation.

For a deeper exploration of Kiali it is recommended to

run through the Kiali Tutorial.

# Cleanup

If you are not planning any follow-up tasks, remove the Bookinfo sample application and Kiali from your cluster.

1. To remove the Bookinfo application, refer to the Bookinfo cleanup instructions.
2. To remove Kiali from a Kubernetes environment:

```
$ kubectl delete -f https://raw.githubusercontent.com/istio/istio/release-1.11/samples/addons/kiali.yaml
```



# Remotely Accessing Telemetry Addons

⌚ 6 minute read ✓ page test

---

This task shows how to configure Istio to expose and access the telemetry addons outside of a cluster.

# Configuring remote access

Remote access to the telemetry addons can be configured in a number of different ways. This task covers two basic access methods: secure (via HTTPS) and insecure (via HTTP). The secure method is *strongly recommended* for any production or sensitive environment. Insecure access is simpler to set up, but will not protect any credentials or data transmitted outside of your cluster.

For both options, first follow these steps:

## 1. Install Istio in your cluster.

To additionally install the telemetry addons, follow the [integrations documentation](#).

## 2. Set up the domain to expose addons. In this example, you expose each addon on a subdomain, such as `grafana.example.com`.

- If you have an existing domain pointing to the external IP address of `istio-ingressgateway` (say `example.com`):

```
$ export INGRESS_DOMAIN="example.com"
```

- If you do not have a domain, you may use

nip.io which will automatically resolve to the IP address provided. This is not recommended for production usage.

```
$ export INGRESS_HOST=$(kubectl -n istio-system get service istio-ingressgateway -o jsonpath='{.status.loadBalancer.ingress[0].ip}')
$ export INGRESS_DOMAIN=${INGRESS_HOST}.nip.io
```

## Option 1: Secure access (HTTPS)

A server certificate is required for secure access.

Follow these steps to install and configure server certificates for a domain that you control.



This option covers securing the transport layer *only*. You should also configure the telemetry addons to require authentication when exposing them externally.

This example uses self-signed certificates, which may not be appropriate for production usages. For these cases, consider using `cert-manager` or other tools to

provision certificates. You may also visit the Securing Gateways with HTTPS task for general information on using HTTPS on the gateway.

1. Set up the certificates. This example uses openssl to self sign.

```
$ CERT_DIR=/tmp/certs
$ mkdir -p ${CERT_DIR}
$ openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 -subj "/O=example Inc./CN=*.${INGRESS_DOMAIN}" -keyout ${CERT_DIR}/ca.key -out ${CERT_DIR}/ca.crt
$ openssl req -out ${CERT_DIR}/cert.csr -newkey rsa:2048 -nodes -keyout ${CERT_DIR}/tls.key -subj "/CN=*.${INGRESS_DOMAIN}/O=example organization"
$ openssl x509 -req -days 365 -CA ${CERT_DIR}/ca.crt -CAkey ${CERT_DIR}/ca.key -set_serial 0 -in ${CERT_DIR}/cert.csr -out ${CERT_DIR}/tls.crt
$ kubectl create -n istio-system secret tls telemetry-gw-cert --key=${CERT_DIR}/tls.key --cert=${CERT_DIR}/tls.crt
```

2. Apply networking configuration for the telemetry addons.
  1. Apply the following configuration to expose

## Grafana:

```
$ cat <<EOF | kubectl apply -f -
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: grafana-gateway
  namespace: istio-system
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 443
      name: https-grafana
      protocol: HTTPS
    tls:
      mode: SIMPLE
      credentialName: telemetry-gw-cert
  hosts:
```

```
- "grafana.\${INGRESS_DOMAIN}"  
---  
apiVersion: networking.istio.io/v1alpha3  
kind: VirtualService  
metadata:  
  name: grafana-vs  
  namespace: istio-system  
spec:  
  hosts:  
    - "grafana.\${INGRESS_DOMAIN}"  
  gateways:  
    - grafana-gateway  
  http:  
    - route:  
        - destination:  
            host: grafana  
            port:  
                number: 3000  
---  
apiVersion: networking.istio.io/v1alpha3
```

```
kind: DestinationRule
metadata:
  name: grafana
  namespace: istio-system
spec:
  host: grafana
  trafficPolicy:
    tls:
      mode: DISABLE
---
EOF
gateway.networking.istio.io/grafana-gateway created
virtualservice.networking.istio.io/grafana-vs created
destinationrule.networking.istio.io/grafana created
```

2. Apply the following configuration to expose Kiali:

```
$ cat <<EOF | kubectl apply -f -
```

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: kiali-gateway
  namespace: istio-system
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 443
        name: https-kiali
        protocol: HTTPS
      tls:
        mode: SIMPLE
        credentialName: telemetry-gw-cert
    hosts:
      - "kiali.${INGRESS_DOMAIN}"
---
apiVersion: networking.istio.io/v1alpha3
```

```
kind: VirtualService
metadata:
  name: kiali-vs
  namespace: istio-system
spec:
  hosts:
    - "kiali.${INGRESS_DOMAIN}"
  gateways:
    - kiali-gateway
  http:
    - route:
        - destination:
            host: kiali
            port:
              number: 20001
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: kiali
```

```
namespace: istio-system
spec:
  host: kiali
  trafficPolicy:
    tls:
      mode: DISABLE
---
EOF
gateway.networking.istio.io/kiali-gateway created
virtualservice.networking.istio.io/kiali-vs created
destinationrule.networking.istio.io/kiali created
```

3. Apply the following configuration to expose Prometheus:

```
$ cat <<EOF | kubectl apply -f -
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
```

```
name: prometheus-gateway
namespace: istio-system
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 443
        name: https-prom
        protocol: HTTPS
      tls:
        mode: SIMPLE
        credentialName: telemetry-gw-cert
    hosts:
      - "prometheus.${INGRESS_DOMAIN}"
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: prometheus-vs
```

```
namespace: istio-system
spec:
  hosts:
    - "prometheus.${INGRESS_DOMAIN}"
  gateways:
    - prometheus-gateway
  http:
    - route:
        - destination:
            host: prometheus
            port:
              number: 9090
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: prometheus
  namespace: istio-system
spec:
  host: prometheus
```

```
trafficPolicy:  
    tls:  
        mode: DISABLE  
---  
EOF  
gateway.networking.istio.io/prometheus-gateway created  
virtualservice.networking.istio.io/prometheus-vs created  
destinationrule.networking.istio.io/prometheus created
```

4. Apply the following configuration to expose the tracing service:

```
$ cat <<EOF | kubectl apply -f -  
apiVersion: networking.istio.io/v1alpha3  
kind: Gateway  
metadata:  
    name: tracing-gateway  
    namespace: istio-system
```

```
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 443
      name: https-tracing
      protocol: HTTPS
    tls:
      mode: SIMPLE
      credentialName: telemetry-gw-cert
  hosts:
  - "tracing.${INGRESS_DOMAIN}"
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: tracing-vs
  namespace: istio-system
spec:
```

```
hosts:
- "tracing.${INGRESS_DOMAIN}"
gateways:
- tracing-gateway
http:
- route:
  - destination:
    host: tracing
    port:
      number: 80
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: tracing
  namespace: istio-system
spec:
  host: tracing
  trafficPolicy:
    tls:
```

```
mode: DISABLE
```

```
---
```

```
EOF
```

```
gateway.networking.istio.io/tracing-gateway created
virtualservice.networking.istio.io/tracing-vs created
destinationrule.networking.istio.io/tracing created
```

### 3. Visit the telemetry addons via your browser.



If you used self signed certificates, your browser will likely mark them as insecure.

- **Kiali:** [https://kiali.\\${INGRESS\\_DOMAIN}](https://kiali.${INGRESS_DOMAIN})

- Prometheus:  
`https://prometheus.${INGRESS_DOMAIN}`
- Grafana: `https://grafana.${INGRESS_DOMAIN}`
- Tracing: `https://tracing.${INGRESS_DOMAIN}`

## **Option 2: Insecure access (HTTP)**

1. Apply networking configuration for the telemetry addons.
  1. Apply the following configuration to expose

## Grafana:

```
$ cat <<EOF | kubectl apply -f -
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: grafana-gateway
  namespace: istio-system
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http-grafana
      protocol: HTTP
    hosts:
    - "grafana.${INGRESS_DOMAIN}"
---
apiVersion: networking.istio.io/v1alpha3
```

```
kind: VirtualService
metadata:
  name: grafana-vs
  namespace: istio-system
spec:
  hosts:
    - "grafana.${INGRESS_DOMAIN}"
  gateways:
    - grafana-gateway
  http:
    - route:
        - destination:
            host: grafana
            port:
              number: 3000
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: grafana
```

```
namespace: istio-system
spec:
  host: grafana
  trafficPolicy:
    tls:
      mode: DISABLE
---
EOF
gateway.networking.istio.io/grafana-gateway created
virtualservice.networking.istio.io/grafana-vs created
destinationrule.networking.istio.io/grafana created
```

2. Apply the following configuration to expose Kiali:

```
$ cat <<EOF | kubectl apply -f -
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
```

```
    name: kiali-gateway
    namespace: istio-system
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 80
        name: http-kiali
        protocol: HTTP
      hosts:
        - "kiali.${INGRESS_DOMAIN}"
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: kiali-vs
  namespace: istio-system
spec:
  hosts:
```

```
- "kiali.${INGRESS_DOMAIN}"  
gateways:  
- kiali-gateway  
http:  
- route:  
  - destination:  
    host: kiali  
    port:  
      number: 20001  
---  
apiVersion: networking.istio.io/v1alpha3  
kind: DestinationRule  
metadata:  
  name: kiali  
  namespace: istio-system  
spec:  
  host: kiali  
  trafficPolicy:  
    tls:  
      mode: DISABLE
```

```
---  
EOF  
gateway.networking.istio.io/kiali-gateway created  
virtualservice.networking.istio.io/kiali-vs created  
destinationrule.networking.istio.io/kiali created
```

3. Apply the following configuration to expose Prometheus:

```
$ cat <<EOF | kubectl apply -f -  
apiVersion: networking.istio.io/v1alpha3  
kind: Gateway  
metadata:  
  name: prometheus-gateway  
  namespace: istio-system  
spec:  
  selector:  
    istio: ingressgateway  
  servers:
```

```
- port:  
    number: 80  
    name: http-prom  
    protocol: HTTP  
  hosts:  
    - "prometheus.${INGRESS_DOMAIN}"  
---  
apiVersion: networking.istio.io/v1alpha3  
kind: VirtualService  
metadata:  
  name: prometheus-vs  
  namespace: istio-system  
spec:  
  hosts:  
    - "prometheus.${INGRESS_DOMAIN}"  
  gateways:  
    - prometheus-gateway  
  http:  
    - route:  
        - destination:
```

```
        host: prometheus
        port:
            number: 9090
    ---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
    name: prometheus
    namespace: istio-system
spec:
    host: prometheus
    trafficPolicy:
        tls:
            mode: DISABLE
    ---
EOF
gateway.networking.istio.io/prometheus-gateway created
virtualservice.networking.istio.io/prometheus-vs created
destinationrule.networking.istio.io/prometheus created
```

4. Apply the following configuration to expose the tracing service:

```
$ cat <<EOF | kubectl apply -f -
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: tracing-gateway
  namespace: istio-system
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http-tracing
      protocol: HTTP
    hosts:
    - "tracing.${INGRESS_DOMAIN}"
```

```
---
```

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: tracing-vs
  namespace: istio-system
spec:
  hosts:
    - "tracing.${INGRESS_DOMAIN}"
  gateways:
    - tracing-gateway
  http:
    - route:
        - destination:
            host: tracing
            port:
              number: 80

```

```
---
```

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
```

```
metadata:
  name: tracing
  namespace: istio-system
spec:
  host: tracing
  trafficPolicy:
    tls:
      mode: DISABLE
---
EOF
gateway.networking.istio.io/tracing-gateway created
virtualservice.networking.istio.io/tracing-vs created
destinationrule.networking.istio.io/tracing created
```

## 2. Visit the telemetry addons via your browser.

- Kiali: [http://kiali.\\${INGRESS\\_DOMAIN}](http://kiali.${INGRESS_DOMAIN})
- Prometheus:  
[http://prometheus.\\${INGRESS\\_DOMAIN}](http://prometheus.${INGRESS_DOMAIN})

- **Grafana:** `http://grafana.${INGRESS_DOMAIN}`
- **Tracing:** `http://tracing.${INGRESS_DOMAIN}`

# Cleanup

- Remove all related Gateways:

```
$ kubectl -n istio-system delete gateway grafana-gateway kiali-gateway prometheus-gateway tracing-gateway
gateway.networking.istio.io "grafana-gateway" deleted
gateway.networking.istio.io "kiali-gateway" deleted
gateway.networking.istio.io "prometheus-gateway" deleted
gateway.networking.istio.io "tracing-gateway" deleted
```

- Remove all related Virtual Services:

```
$ kubectl -n istio-system delete virtualservice grafana-vs  
kiali-vs prometheus-vs tracing-vs  
virtualservice.networking.istio.io "grafana-vs" deleted  
virtualservice.networking.istio.io "kiali-vs" deleted  
virtualservice.networking.istio.io "prometheus-vs" deleted  
virtualservice.networking.istio.io "tracing-vs" deleted
```

- Remove all related Destination Rules:

```
$ kubectl -n istio-system delete destinationrule grafana ki  
ali prometheus tracing  
destinationrule.networking.istio.io "grafana" deleted  
destinationrule.networking.istio.io "kiali" deleted  
destinationrule.networking.istio.io "prometheus" deleted  
destinationrule.networking.istio.io "tracing" deleted
```